Table of contents

Introduction

This course presents the basic facts in the theory of partial recursive functions. The investigation will be continued in the course *Decidability and Complexity Classes*. In what follows the present course will be referred to as (CT), its sequel by (DCC).

Roughly speaking, the whole area of algorithm theory can be divided into two parts, which are referred to as *design and analysis of algorithms* (*DAA*), and *theory of computation* (*TC*). DAA deals with the problem of finding efficient algorithms for specific problems, like sorting, integer factorization etc., the complexity analysis of such algorithms, and general principles concerning their construction, like dynamic programming, recursion etc. Clearly, in this context a problem normally is not an individual mathematical question, like 'Is Goldbach's conjecture true?,' but a class C of related questions, like 'Factor n, for any given number n.' Every input n corresponds to an *instance* of the problem, that is, to an element of C.

It turns out that not every problem can be solved by an efficient algorithm. Specifically, the following facts have been shown:

(a) Some mathematical problems are *algorithmically undecidable*, that is, they cannot be solved algorithmically, even in principle. An example is the following: Does a given program halt for every input, or does it sometimes get into an infinite loop? Results like this hold for every common programming languages. In chapter 5 the algorithmic undecidability of this problem and several related ones will be shown for the primitive programming language called $\mathcal{WL}$, on which this lecture is based.

(b) There are mathematical problems for which algorithmic solutions exist, but for which every algorithm requires an enormous amount of time and space (in relation to the size of the input). An extreme example is the *containment problems for finite reachability sets*, defined in *Cardoza, Lipton and Meyer: Exponential Space Complete Problems for Petri Nets and Commutative Semigroups, Proceedings of the Eighth Annual Symposium on the Theory of Computing, 1976.* This problem will not be dealt with in (CT) and (DCC). A less extreme, but more well-known example is the decision problem for the elementary theory of the structure $(\mathbb{Z}, +, \leq, 0)$ (Presburger arithmetic).

Such results have of course important consequences. For instance, when designing a compiler, it would be very useful to include a check for possible infinite loops. However, result (a) implies that no such check is possible, at least none that works for *every* program.

How is it possible to prove results like (a), (b)? For this purpose one must first of all define a mathematical equivalent for the intuitive notion of algorithmically computable function. The concept of partial recursive function, defined in chapter 1, is generally accepted as such an equivalent, see the discussion about Church's thesis (ChT) in section 3.4. From now on we use the word *'computable'* (in quotes) for *algorithmically computable in the intuitive sense*; the quotes indicate that this is not a mathematically precise concept.

After providing the basic definitions one has to carry out a systematic investigation of the class of partial recursive functions, of decidable and undecidable problems, the complexity of algorithms in general (not just for specific practical problems) etc. All this is the subject of TC, dealt with in (CT) and (DCC).

In (CT) the notion of partial recursive function, and related notions like that of recursively enumerable set etc., are introduced and investigated. In the first half of (DCC) more results in the area of recursive function theory are presented, and the undecidability of various problems outside this area is shown as well. The second half of (DCC) deals with abstract computational complexity theory. A typical question in this area is: If f and g are positive functions such that g grows faster than f, does there necessarily exist some problem which can be solved within complexity bound g, but not within complexity bound f? (A precise statement of this question actually requires some of the concepts to be introduced.)

Of course, TC is much more theoretical and academic than DAA. Nevertheless, the undecidability results mentioned show that TC, too, is relevant for applications. This is even more apparent in the theory of polynomial time-complexity. Here certain problems of considerable practical importance are dealt with (like the travelling salesman problem), but within a theoretical framework which provides insights that could not be achieved otherwise. Various concepts and proofs in this area have analogues in recursive function theory, and a comparison again yields important insights.

Outline of these lecture notes. In chapter 1 the basic concepts are introduced: primitive recursive, recursive and partial recursive functions, (primitive) recursive and recursively enumerable sets and predicates. The definitions are based on a very simple programming language.

In chapter 2 it is shown that many 'computable' functions which occur in practice actually belong to the class of primitive recursive functions (the narrowest of the three classes of functions), and that all these classes of functions and sets are closed under many operations.

Chapter 3 contains the first collection of results which are typical for recursive function theory and provide insight into the structure of the classes of functions and sets introduced in chapter 1. Most fundamental among these results are the existence of partial recursive *universal* functions and the S-m-n-theorem. Diagonalization is introduced as a fundamental method for showing certain results like the undecidability of the halting problem. The chapter concludes with a discussion of ChT, which asserts that the recursive functions are exactly those functions which can be computed by some algorithm in the intuitive sense.

Chapter 4 introduces alternative models of computation, that is, alternative ways to define the class of partial recursive functions. Three of them (μ-recursion, Turing machines and Markov algorithms) are shown to be equivalent to the formalism from chapter 1. Other models investigated in literature are mentioned, but not treated in detail.

In chapter 5 the systematic investigation of the class of partial recursive functions and of recursively enumerable sets is continued, and another fundamental proof method (proof by reduction) is introduced. It contains a collection of topics, as indicated by the title of that chapter.

In chapter 6, finally, reducibility relations and degrees of unsolvability are treated systematically, a topic which will be covered in more detail in (DCC).

It can be expected that the lecture itself will follow the lecture notes rather closely, except that the proofs of some theorems and other details will be omitted. For instance, the proof of theorem 1.1 will be presented partially at most. The lecture notes, on the other hand, do contain all such details, thus serving as a reference.

A few exercises have been interspersed in the text; in some cases they cover details omitted in the main part of the text. A rating by difficulty, on a scale from 0 to 8, is given in brackets, for instance, exercise 1.1 ... (i) [4]. We describe roughly what each even rating means; odd ratings indicate an intermediate difficulty, of course.

[0]: (Triviality) Such an exercise should be solved almost instantly, if the material has been understood.

[2]: (Ordinary exercise) Such an exercise is still fairly simple and straightforward, but the solution may require a few lines to write down and/or some moments of thought. Exercises that would be suitable for written exams typically have this rating. (But the converse is not true: Not every exercise with rating [2] is suitable for a written exam!)

[4]: (Puzzle) Such exercises require non-obvious tricks, and the solution may be somewhat longer than for exercises rated [0] or [2]. Textbooks typically mark problems of this difficulty by a single star. Still, with some patience, students should be able to find a solution. The time it takes depends of course on how quickly one hits upon the right idea, but 15–45 minutes should be reasonable.

[6]: (Challenge) Such problems are definitively not trivial, and the solution requires methodical work. Something like two hours should be a reasonable time to do such an exercise.

[8]: (Research problem) In this case it will be very hard for students to find a solution, unless they have already some experience in the area of recursive function theory, or make a considerable effort over several days at least, or study literature that goes beyond the course. These problems should be considered as a stimulation for serious work in the area of recursive function theory.

Naturally, the rating is only a guideline and cannot be expected to provide an acurate indication of the actual effort required. It refers to the difficulty of the exercise based on the material covered up to that point, including results of earlier exercises; some exercises would be easier if they were given at a later stage.

Prerequisites. The theory of computation requires very little specific knowledge from other areas of mathematics, but it does require mathematical maturity. The student should have at least a superficial knowledge of some Algol-like programming language, and he/she is supposed to be familiar with the basics of mathematical logic (predicates, quantifiers etc.) and of set theory (set, function, intersection etc.); but no special results in these areas are needed.

Notation. In general the notation used in this course follows the usual mathematical conventions. Some points, where the use of symbols is not uniform in literature, must be mentioned, though. $A \subseteq B$ means that A is a subset of B, but not necessarily a proper one, whereas $\subset$ and $\subsetneq$ both denote the *proper* subset relation. The notation $A \subset B$ is used if the primary emphasis is on $A \subseteq B$, while $A \neq B$ may be obvious (or it may have been established earlier). $|A|$ is the cardinality (number of elements) of the set A, 2^A its power set.

In these lecture notes, as in most papers on recursive function theory, ω denotes the set of natural numbers including 0. Most sets considered will be subsets of ω^n for some n, and similarly, functions will usually have domain and range contained in ω^n and ω respectively. Therefore we introduce the symbols $\mathcal{NS}$ and $\mathcal{NF}$ to denote the class of all such sets and functions respectively. If ${}^{A}B$ denotes the set of all functions from A to B, then

$$\mathcal{NS} = \bigcup_{n=0}^{\infty} \omega^n,$$
$$\mathcal{NF} = \bigcup_{n=0}^{\infty} {}^{\omega^n}\omega.$$

The notation ${}^{A}B$ will not be needed later.

The logical *and* is denoted by & rather than $\wedge$. As usual, logical equivalence is denoted by $\iff$; however, we write $P(x) \equiv \phi(x)$, if this formula defines the predicate P or provides a representation of P to show that P is (primitive) recursive; see, for instance, example 2.4 (v).

Domain and range of a function f are denoted by $\operatorname{dom} f$ and $\operatorname{ran} f$, respectively. When a function $f: A \to B$ is introduced, then it is not assumed that $\operatorname{dom} f$ is necessarily all of A, unless it is explicitly stated that f is total. The symbol $f(x)\downarrow$ means that $f(x)$ is defined (converges), whereas $f(x)\uparrow$ means that $f(x)$ is undefined (diverges). $\vec{x}_n$ stands for the vector $(x_1, \ldots, x_n)$, or just for '$x_1, \ldots, x_n$,' see below. Finally, the lambda-notation is used throughout; thus $\lambda xy.t$ means: the value t, regarded as a function of x and y. It also applies if t is not defined for all values of x and y; in this case the function is not total. Also, the number of arguments may be 0; thus $\lambda.t$ denotes the function with 0 arguments whose value is t. This notation is also used for predicates: $\lambda xy.\phi$ is the assertion ϕ, regarded as a predicate about x and y.

Additional symbols for concepts to be defined, but also for specific functions, sets etc. will be introduced later. Sometimes it may not be obvious from the context whether the meaning of a certain symbol is permanent or just temporary, that is, whether it applies throughout the lecture notes, or only to the current example or exercise etc. In such cases a permanent meaning is indicated by preceding the symbol by (P), as in example 2.5, for instance. It is clear that symbols introduced under **Definiton** or **Notation**, and some symbols introduced at other places (as in the previous paragraphs) also have a permanent meaning; the marking (P) is only applied in case of doubt.

Conventions. Unless otherwise stated, the range of quantifiers is assumed to be ω^n in formulas like $(\forall \vec{x}_n)(\ldots)$ (and similarly with other letters instead of x, n). If $A \subseteq \omega^n$, then $\overline{A}$ denotes the complement of A with respect to ω^n. The phrase 'an assertion holds for almost all $\vec{x}_n$,' means that it holds for all $\vec{x}_n \in \omega^n$ with at most finitely many exceptions.

Whenever some property is defined for a set $A \subseteq \omega^n$, then it is also applied to the corresponding predicate $P = \lambda\vec{x}_n.\vec{x}_n \in A$. Thus $P \in \mathcal{NS}$ means the same as $A \in \mathcal{NS}$, and predicates may be infinite, recursive etc. (The latter property will be introduced in definition 1.6 (ii)). Likewise, set-theoretic notation may be applied to predicates; for instance, if P and Q are predicates over ω^n, then $P \cup Q$ denotes the predicate $\lambda\vec{x}_n.P(\vec{x}_n) \vee Q(\vec{x}_n)$, which corresponds to the union of the sets corresponding to P and Q respectively.

Similarly, a function f corresponds to the set (P) $\operatorname{gr}(f) = \{\, (\vec{x}_n, y) \mid f(\vec{x}_n) = y \,\}$, which is called the *graph* of f. Some terminology will be based on this correspondence: f is called finite (infinite) iff $\operatorname{gr}(f)$ (or, equivalently, $\operatorname{dom} f$) is finite (infinite). Likewise, $f \subseteq g$ means $\operatorname{gr}(f) \subseteq \operatorname{gr}(g)$; this is equivalent to the assertion that g is an *extension* of f, that is, $g(x) = f(x)$ whenever $f(x)\downarrow$. Occasionally, however, a function has to be distinguished from its graph; see example 2.3 (iv) and theorem 3.14. This is due to the fact that the terms '(primitive) recursive function' and '(primitive) recursive set' are defined separately. Therefore we do *not* adopt the point of view often taken in set theory, where a function f is defined to be a certain set of pairs, namely just $\mathrm{gr}(f)$.

In the previous paragraph the tuple $(\vec{x}_n, y)$ occurred; it means $(x_1, \ldots, x_n, y)$, not $\big((x_1, \ldots, x_n), y\big)$. In this context, and also in formulas like $f(\vec{x}_n) = y$, $\vec{x}_n$ is just an abbreviation for '$x_1, \ldots, x_n$' (without parenthesis), whereas in formulas like $\vec{x}_n \in A$ it is an abbreviation for $(x_1, \ldots, x_n)$. No confusion will result from this slight ambiguity.

When new concepts are introduced, we wish to indicate to what extent our definitions are standard, that is, agree with those in literature. The default assumption is that our definition does indeed agree with most textbooks. If not, then we attach (N), (V) or (E), the meaning being as follows:

(N): The term defined is used only in these lecture notes, not in other textbooks.

(V): The meaning of the term defined varies slightly from textbook to textbook and often differs from ours (though not in a very essential way). In definition 3.2, for instance, there are infinitely many appropriate enumerations of the partial recursive functions, each serving the same purpose; it is unlikely that any textbook uses precisely the same as ours.

(E): The meaning of the term defined is standard, but many textbooks define it in a different, though equivalent way. For instance, the partial recursive functions (definition 1.5) can also be introduced by any of the models of computation from chapter 4.

1. The basic concepts

1.1. Algorithms and procedures.

In this section we discuss some basic *informal* concepts of TC: *problems*, *algorithms* and *procedures*. The first two were already used in the introduction, but now a more careful explanation is in order.

Recall that a *problem* P is a class of related questions, and that each of these questions is an *instance* of P. In addition it must be possible to describe every instance of P and the answer for each instance by a finite expression in some language. For instance, consider the following directions:

(i) 'Decide if x is rational, for any given real number x.'

(ii) 'Compute the real solutions of $ax^2 + bx + c = 0$ for given integers a, b, c.'

Most real numbers have no finite representation, and it makes no sense to consider a question that cannot be written down as a finite English sentence or mathematical formula. Therefore (i) does not qualify as a problem. Similarly, in example (ii) infinite decimal fractions cannot be required as answers. The solutions may instead be represented either approximately (by finite decimal fractions), or symbolically (using the square root symbol). For a complete specification of the problem one has to tell which form of the output is asked for.

There are two main types of problem. One is the computation of functions: find $f(x)$ for given x. The second kind of problem occurs when the answer for each instance is 'yes' or 'no,' for instance: Given an integer $x > 0$, decide if x is a prime number or not. This may be regarded as a special case of the first type of problem, where the range of the function to be computed is $\{$'yes','no'$\}$, or perhaps $\{1, 0\}$, if 1 and 0 are interpreted as codes for 'yes' and 'no.'

Intuitively, an *algorithm* $\mathcal{A}$ *for a problem* P is a set of instructions which allows to solve any given instance of P. The *input* for $\mathcal{A}$ characterizes the instance of P to be solved, and the *output* is the corresponding answer. The execution of $\mathcal{A}$ has to terminate for every input. There are certain additional criteria an algorithm has to satisfy; they will be given in definition 1.1 below. We emphasize that the notion of algorithm is not a rigorous mathematical one, like that of a group. Therefore it does not suffice to provide a definition; some further discussion will be needed for clarification.

Following McNaughthon [15] and other textbooks, we use the term *procedure* for a set of instructions which satisfies similar conditions as an algorithm, except that it does not necessarily terminate for every input. As it turns out, algorithms cannot be investigated in general without studying procedures as well. As we noted in the introduction, there is no way to decide whether or not a given program P always terminates, that is, whether P represents an algorithm or just a procedure. Hence the class of programs (of the programming language under consideration) has to be studied as a whole. The same remark also applies to every other representation of algorithms.

There is yet another reason why procedures other than algorithms are considered: TC considers certain computational tasks (other than the solution of problems in the above sense) which by their nature cannot be performed in a finite amount of time. For instance, it is easy to design a procedure for generating all prime numbers, or for computing all digits of π. Such a procedure will run forever, and from time to time it will give a new prime number, or a new digit of π, as output.

Another important task performed by procedures rather than algorithms is the computation of *partial* functions. Let $\mathcal{A}$ be a procedure which takes an element of a set A as input and yields at most one element of B as output. For some inputs $\mathcal{A}$ may not yield any output, possibly by running forever. Then it is natural to associate with $\mathcal{A}$ the partial function $f\colon A \to B$ defined by

$$f(x) = \begin{cases} y & \text{if } \mathcal{A} \text{ yields output } y \text{ for input } x, \\ \uparrow & \text{if } \mathcal{A} \text{ yields no output for input } x; \end{cases}$$

we say that f is the partial function computed by $\mathcal{A}$. A given partial function f is called 'computable' iff it is computed by some procedure $\mathcal{A}$ in the sense described. Clearly, $\mathcal{A}$ is an algorithm if and only if f is total.

If f is not total, then it may be argued that $\mathcal{A}$ does not compute f in a satisfactory manner. Indeed, if the execution of $\mathcal{A}$ for input x neither terminates nor yields an output, then it does not decide if $f(x)\downarrow$. However, this defect is sometimes unavoidable, since membership in $\operatorname{dom} f$ may be undecidable, see definition 1.7 and theorem 3.15. We are now ready for a semi-formal definition of *algorithm* and *procedure*. It is essentially the same as the one given in McNaughthon [15], p. 4. Other textbooks provide similar definitions, see, for instance, Knuth [11], p. 4–6, or Rogers [21], p. 2.

Definition 1.1: (V) (i) An *algorithm* $\mathcal{A}$ *for a problem* P is an organized set of instructions for solving any instance of P, such that

- (A1) $\mathcal{A}$ has a *finite representation* in some language.
- (A2) At the beginning or during the execution $\mathcal{A}$ takes as *input* certain values which characterize the instance of P to be solved. (No input is needed if P has only one instance.)
- (A3) The answer for the instance of P to be solved is given by $\mathcal{A}$ as *output*, either explicitly during the computation, or as a well-defined part of the final result.
- (A4) The execution itself proceeds in a *stepwise* fashion, and the task performed at each step is *simple*. At least each step must be simple enough to be readily performed by a computer or human, without any particular creative effort.
- (A5) $\mathcal{A}$ must be *deterministic*, that is, the action performed at each step must be rigorously and unambiguously specified.
- (A6) The execution of $\mathcal{A}$ must *terminate* after a finite number of steps for each input.

(ii) A *procedure* is an organized set of instructions satisfying requirements like an algorithm, except that it does not necessarily terminate for every input. Also, it may yield finitely or infinitely many output values or no output at all, according to the requirements of the task performed.

No comment is needed on condition (A1). The language used may be ordinary English or some programming language, and there are other possibilities as well.

As for conditions (A2) and (A3), we note first that some textbooks require that the input enters $\mathcal{A}$ *before* execution, and that the output appears as a part of the final result. In this case procedures with infinite output are ruled out. For procedures with finite output this requirement does not constitute a very serious restriction, and in fact it will be satisfied by most algorithms and procedures presented here. Nevertheless, there is no need to include it in definition 1.1; for instance, computer programs may contain input- and output commands anywhere.

In condition (A2) we also allow the number of input values to be zero. Indeed, there are problems without input, for instance, the problem of computing the first 10^8 decimal digits of π. True, this is not a very practical example; normally the number of digits to be computed would be taken as an input value. A more relevant example occurred in connection with the famous four-color problem:

($*$) Is it possible to color the nodes of every planar graph such that no two adjacent nodes receive the same color, and such that no more than four colors are used?

The crucial theoretical breakthrough occurred when it was found that the answer could be determined in a finite number of steps. Thus an algorithm for deciding the question ($*$) was known at that stage; its implementation still required considerable effort. Of course, such an algorithm has to be executed only once, since only one fixed question has to be answered.

According to our definition, every algorithm must yield an output for every input, but this requirement does not apply to procedures. Indeed, an algorithm which sometimes yields no output does not make sense, for in this case it does not answer an instance of the problem to be solved. By contrast, it was already noted that procedures for computing nontotal functions yield an output only for some, not for all inputs. We may even consider procedures which *never* yield an output, and which perform no particular task. Such procedures serve no practical purpose of course, but in purely theoretical investigations it may be of interest to study their complexity behaviour, or related questions. As an example, we mention the following open problem: Does the following procedure (without output!) terminate for every input $N \in \omega$?

```
while N > 1 do
    if odd(N)
        then N := 3 * N + 1
        else N := N/2
```

For instance, if the input is 3, then N takes the values 3, 10, 5, 16, 8, 4, 2 and 1 in turn, whereupon the program terminates.

(A4) is probably the vaguest of the conditions (A1)–(A6), and the one which is hardest to specify mathematically. The requirement that an algorithm $\mathcal{A}$ must be capable of being performed by a human or by some computing device is called *effectiveness*. This term applies to the algorithm as a whole as well as to the individual steps. Basic operations like additions obviously can be carried out effectively. An example of an instruction where this is not clear is the following conditional: 'If there exist integers y and z such that $y^3 - z^2 = u$, then set x to 1, else set x to 0.' It is not possible to check the condition $y^3 - z^2 = u$ for all pairs (y, z). The vagueness in (A4) disappears as soon as some particular formalism for describing algorithms is chosen, for then it is also specified precisely what one *step* of a computation is. However, such a definition can hardly account for everything that might be a step of an algorithm (in the intuitive sense). What one can hope for is that it leads to an appropriate definition of 'computable function,' and this is just what ChT is about.

One consequence of condition (A4) is that the result of each step must have a finite representation; this is just the requirement which also applies to the algorithm or procedure $\mathcal{A}$ itself, and to the input and output. Indeed, it is principally impossible to perform a step whose result involves an infinite amount of information.

Condition (A5) implies that any two executions of $\mathcal{A}$ for the same input will be essentially identical, independently of the human or computing device who performs it. Of course, two such executions may differ in certain respects which play no essential rôle. For instance, a person may use a blackboard or a sheet of paper, a computer will probably be faster, etc.

When an algorithm $\mathcal{A}$ is presented in a textbook, it often contains operations which in turn can be performed in various ways. Consider, for instance, the instruction $X := Y * Z$. Multiplication can be done either by the elementary school-algorithm, or by a more refined algorithm which works faster for large Y and Z. On an even higher level an instruction may be something like 'Sort the array A,' or 'Factor the number n.' In connection with condition (A5), such instructions may be viewed in two different ways. Operations like multiplications may be considered basic, ignoring the fact that they are performed in even smaller steps. Then two executions of $\mathcal{A}$ which differ only in the way the operation in question is performed are still considered 'essentially identical.' This point of view is sometimes, but not always, appropriate. Indeed, when the complexity of $\mathcal{A}$ is investigated, one has to be aware that the amount of time required for a multiplication depends on the size of the numbers involved and on the algorithm used. Alternatively, it may be understood that the given description of the algorithm is incomplete, and that a complete description would have to include directions for performing the operation under consideration. In this case we say that the algorithm $\mathcal{A}$ is *incompletely specified*. Of course, there are important reasons why a complete description is often omitted: the goal may be to show the main idea of $\mathcal{A}$ without considering details.

Another kind of instruction which seems to violate condition (A5) even more clearly is one which involves random choices, for instance: 'Assign to x any element of the finite set F.' Such an instruction may again be viewed in different ways, depending on the reason for its occurrence. For one thing it may be irrelevant which element of F is assigned to x, since every choice works. In this case we have just another example of an incompletely specified operation; a complete description of the algorithm is obtained simply by replacing the random choice by some deterministic one, like 'Assign to x the smallest element of F.'

Secondly, there are algorithms of the Monte Carlo or Las Vegas type. Here the crucial feature is as follows: under the assumption that random choices are truely random it can be shown that the algorithm $\mathcal{A}$ yields the right result with a certain (high) probability, or that it has a certain expected running time. Such algorithms do not fall under definition 1.1, and in fact, they will not be dealt with in these lecture notes any further. True, a deterministic algorithm $\mathcal{A}'$ is obtained if the 'random' choices are made by computing pseudo-random numbers. Conceptually, however, $\mathcal{A}'$ cannot be regarded as equivalent to $\mathcal{A}$. The reason is that probabilistic assertions about the behaviour of $\mathcal{A}'$ do not make sense from an objective (mathematical) point of view, although for humans with incomplete knowledge they still provide useful information.

Finally, there exists the purely theoretical concept of nondeterministic algorithm. It will be mentioned briefly in section 4.5, and it will play a crucial rôle in (DCC), namely in the theory of $\mathcal{NP}$-completeness. At the moment we just note that this concept is once again an extension of, but not contained in definition 1.1.

The first attempts to formalize the informal notions of *algorithm* and *computable function* were made in the 1930's; others followed in subsequent decades. There are some features which all of them have in common. In each case a formalism is introduced which allows to define a class $\mathcal{P}$ of 'computable' partial functions, either directly or by defining a corresponding class of algorithms; the elements of $\mathcal{P}$ are called the *partial recursive functions*. By definition $\mathcal{P}$ is contained in $\mathcal{NF}$, that is, every partial recursive function is of the form $f: \omega^n \to \omega$ for some n. Actually, some formalisms define the concept of computability first for functions $f: (\Sigma^*)^n \to \Sigma^*$, where Σ^* is the set of strings over an alphabet Σ, see section 2.3. However, by associating with every string $w \in \Sigma^*$ a *code number* $\mathrm{cd}_{\Sigma^*}(w) \in \omega$, one obtains nevertheless a class of corresponding functions $g: \omega^n \to \omega$, that is, a definition of $\mathcal{P} \subseteq \mathcal{NF}$.

The process of encoding various types of mathematical objects by natural numbers is called *arithmetization*. It allows to formalize the concept of 'computability' for functions on mathematical objects like graphs, polynomials, finite sets etc. The coding scheme has to be defined anew for each data type. It could be made uniform (leading to a uniform concept of computability for arbitrary mathematical objects) only if a uniform representation of such objects (as strings or sets, say) were given.

It turns out that all formalisms for defining computability are equivalent in the sense that they yield the same class of partial recursive functions $\mathcal{P}$. All functions in $\mathcal{P}$ are obviously 'computable' (under each definition of $\mathcal{P}$). However, the converse assertion (for functions in $\mathcal{NF}$) is not at all obvious; as noted, this is the contents of ChT. Hence the appropriateness of any given definition of $\mathcal{P}$ cannot be expected to be immediately apparent. Most people will agree or disagree with ChT only after studying the set $\mathcal{P}$ for some time.

Another common feature of most formalisms for defining $\mathcal{P}$ is *simplicity*. For instance, definition 1.5 characterizes $\mathcal{P}$ as the set of functions computable by programs of a very primitive programming language with a very restricted set of instructions. Later on it will be shown that functions computed by programs of a more powerful language or constructed by certain definition schemes are still partial recursive. Why do we not use a high-level programming language in the first place? This would be appropriate if we were only interested in providing convenient programming tools and proving specific functions to be partial recursive. But in TC it is even more important to prove structural properties of the set $\mathcal{P}$, and to show that certain functions are not partial recursive etc., and this can be accomplished more easily if the definition of $\mathcal{P}$ is simple. Moreover, it is of theoretical interest to determine the smallest instruction set for a programming language which is sufficient to compute all partial recursive functions.

We note that similar remarks apply for mathematical logic. There are many logical systems (consisting of logical axioms and rules of inference) all of which lead to the same concept of provability. Each system is easily seen to be *correct*, that is, it allows to derive a theorem ϕ from a theorem ψ only if ϕ really follows from ψ. The converse of this fact is not obvious, but in contrast to ChT it can be shown mathematically (Gödel's completeness theorem). This is due to the fact that an assertion like 'sentence ϕ follows from ψ' does have a precise mathematical meaning (semantic implication). The goal is not only to use the rules of inference for mathematical proofs, but also to show assertions about the logical system, like the non-provability of certain theorems. For this purpose the rules of inference initially introduced are kept as simple as possible; afterwards it is shown that they can be used to carry out more complicated deductions as well.

1.2. The language $\mathcal{WL}$.

In this section we introduce a simple programming language which will be used later to define the class of partial recursive functions and the narrower classes of *recursive* and *primitive recursive* functions. We wish to make $\mathcal{WL}$ as simple as possible, yet powerful enough so that all constructs of common high-level programming languages can be simulated. It turns out that assignments and **while**-loops, both of a very restricted form, are sufficient for this purpose. The name $\mathcal{WL}$, which is an abbreviation for while-language, is inspired by the fact that this language contains certain **while**-loops as the only control structure, and in fact it is similar to the while-language introduced in McNaughthon [15], p. 52ff.

Here is an example of a $\mathcal{WL}$-program:

Example 1.1: The following program contains all constructs of $\mathcal{WL}$, despite its extreme simplicity:

0 **while** $X > 0$ **do**;
1 $X := X \dot{-} 1$;
2 $Y := Y + 1$;
3 **end**

The line numbers at the left are not part of this program; they only serve for later reference.

The statement **end** delimits the range of the **while**-loop. In general **while**- and **end**-statements have to match properly, see definition 1.2 (v). The semantics of this program is obvious, except that the symbol $\dot{-}$ has to be explained. It denotes a binary operation called *modified subtraction*, which is defined as follows:

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y, \\ 0 & \text{if } x < y. \end{cases}$$

The reason why modified rather than ordinary subtraction is used is that all partial recursive functions belong to $\mathcal{NF}$ by definition (see the notation introduced in the introduction), and the values of program variables are also restricted to elements of ω. If at some stage during a computation X has value 0, then the instruction $X := X - 1$ could not be executed.

It is easy to see what the program P from example 1.1 accomplishes. Let x and y be the initial values of X and Y, respectively. Then the loop is executed x times, after which the computation terminates. At this point Y has the value $x + y$. Using the terminology from below, we say that program P computes $\lambda xy.x+y$ with respect to X, Y and Y.

An *unbounded register machine* (*URM*) U is characterized by a set R of registers and a program P. It may be thought of as an abstract computing device capable of executing the one fixed program P. The value of each variable of P is stored in some register, and there may be some additional registers which do not correspond to any variable of P. If A is such a register, then the fact that A is never changed is part of the description of the computation effected by U. The word *unbounded* indicates that the values stored in the registers may be arbitrarily large. From now on we will use the terms *variable* and *register* interchangeably.

We are now ready for a formal definition of the syntax of $\mathcal{WL}$. To the author's knowledge the name $\mathcal{WL}$ has not been used in other textbooks, but our definition of *unbounded register machine* is virtually identical to the one given in Shepherdson/Sturgis [22].

Definition 1.2: (V) (i) An *identifier* is a string of capital letters and digits beginning with a letter.

(ii) A $\mathcal{WL}$*-assignment* is an instruction of the form $X := X + 1$ or $X := X \dot{-} 1$, where X is an identifier.

(iii) A *(restricted)* **while**-*statememt* is a statement of the form **while** $X > 0$ **do**, where X is an identifier; an **end**-*statement* is simply a statement of the form **end**.

(iv) An $\mathcal{WL}$-statement is either a $\mathcal{WL}$-assignment, or a restricted **while**-statement, or an **end**-statement.

(v) A $\mathcal{WL}$-program is a (possibly empty) sequence of $\mathcal{WL}$-statements separated by semicolons, with **while**- and **end**-statements properly matched. The latter condition can be formalized as follows. Let $I_0, \ldots, I_{l-1}$ be the sequence of $\mathcal{WL}$-statements in question. Then it forms a $\mathcal{WL}$-program iff there exists a *matching function* $r\colon \{0, \ldots, l-1\} \to \{0, \ldots, l-1\}$ such that

  (a) If I_i is a **while**-statement, then $I_{r(i)}$ is an **end**-statement and $r(i) > i$; otherwise $r(i)$ is undefined;

  (b) If I_j is an **end**-statement, then $j = r(i)$ for some **while**-statement I_i;

  (c) If I_i and I_j are **while**-statements and $i < j < r(i)$, then $r(j) < r(i)$. This implies in particular that r is one-one.

If I_i is a **while**-statement, then I_i *matches* $I_{r(i)}$ and vice versa, and the sequence of statements between I_i and $I_{r(i)}$ is the *body* of I_i (see exercise 1.1 (ii)).

(vi) The *variables* of a $\mathcal{WL}$-program P are the identifiers which appear in the rôle of X in the statements of P as represented in (ii) and (iii).

(vii) An *unbounded register machine (URM)* U is a pair (R, P) such that R is a set of identifiers, P is a $\mathcal{WL}$-program, and R contains all variables of P. R is called the set of *registers* of U.

As noted in the introduction, there exists no algorithm for deciding if a given program (in any common programming language) halts for every input; this assertion holds for $\mathcal{WL}$-programs as well. In particular, no syntactical criterion allows to single out those $\mathcal{WL}$-programs which compute total functions. The next definition does, however, single out a class of $\mathcal{WL}$-programs which contains some (but not all) programs computing total functions:

Definition 1.3: (N) A $\mathcal{WL}$-program P is *special*, if the following holds for every **while**-statement I of P: if X is the identifier occurring in I (that is, I is of the form **while** $X > 0$ **do**), then the statement immediately following I is $X := X \dot{-} 1$, and the body of I contains no other assignments to X.

Example 1.2: The program from example 1.1 is special.

Intuitively it is clear that special $\mathcal{WL}$-programs compute only total functions, since every execution of the body of a **while**-statement **while** $X > 0$ **do** decreases the value of X by 1, until X has value 0. A rigorous formal proof of this fact will be given in theorem 1.2. The functions computed by special $\mathcal{WL}$-programs form an interesting and important class, which will be introduced in definition 1.5.

Exercise 1.1: The task of this exercise is to establish certain facts related to the parsing of $\mathcal{WL}$-programs. The fact that the matching between **while**- and **end**-statements is uniquely determined is of course well-known for other pairs of syntactic items that have to match properly, like parenthesis; here this is formally derived from the properties of r.

Let $P = I_0; \ldots; I_{l-1}$ be a sequence of $\mathcal{WL}$-statements separated by semicolons. For $i \leq l$ let $\#_1(i)$ and $\#_2(i)$ be the number of **while**- and the number of **end**-statements among $I_0, \ldots, I_{i-1}$ respectively, and let $d(i) = \#_1(i) - \#_2(i)$. (If $i = 0$, then $\#_1(i)$, $\#_2(i)$ and $d(i)$ are all 0.) Prove the following facts:

(i) [4] P is a $\mathcal{WL}$-program if and only if $d(l) = 0$ and $d(i) \geq 0$ for all $i \leq l$.

For the remaining parts assume that P is indeed a $\mathcal{WL}$-program and that r is a matching function for P as defined in definition 1.2 (v).

(ii) [4] If I_i is a **while**-statement, then $r(i)$ is the smallest $j > i$ such that $d(j+1) = d(i)$. Hence r is uniquely determined, and so is the match of any given **while**- or **end**-statement.

(iii) [3] One and only one of the following cases holds:

(α) P is empty;

(β) P consists of a single assignment statement;

(γ) P is of the form $P_1; P_2$, where P_1 and P_2 are nonempty $\mathcal{WL}$-programs;

(δ) P is of the form **while** $X > 0$; P_1; **end**, where P_1 is a $\mathcal{WL}$-program.

Conversely, every sequence of statements of one of the forms (α)–(δ) is a $\mathcal{WL}$-program.

(iv) [3] Let $P_1 = I_{l_0}; \ldots; I_{l_1-1}$, where $0 \leq l_0 < l_1 \leq l$. Then P_1 is a $\mathcal{WL}$-program if and only if for every **while**-statement I_i of P we have $l_0 \leq i < l_1 \iff l_0 \leq r(i) < l_1$. If this condition is satisfied and r_1 is the matching function for P_1, then $r(i+l_0) = r_1(i) + l_0$ for $i < l_1 - l_0$.

(v) [2] If P is of the form (γ) from part (iii), then P is special if and only if P_1 and P_2 are special. If P is of the form (δ), then P is special if and only if P_1 is special, begins with the statement $X := X \dot{-} 1$, and contains no other assignments to X.

Remark 1.1: Certain symbols like X are used both as specific identifiers and as metavariables representing arbitrary identifiers. The first kind of use occurs primarily in examples, like example 1.1, the second in most other places, for instance in various parts of definition 1.2. In general it should be clear from the context in which sense X or Y etc. is applied.

So far the *semantics* of $\mathcal{WL}$ has been dealt with only in a rather informal way. For a rigorous mathematical treatment it is necessary to give a formal definition of the *function computed by a $\mathcal{WL}$-program.* Two approaches are possible, not only for $\mathcal{WL}$-programs, but also for programming languages in general. The first, *operational semantics*, uses the concept of *configuration* or *instantaneous description*, which will also be defined for Turing machines and Markov algorithms in chapter 4. Intuitively, a configuration is a snapshot, that is, a mathematical object which contains all the relevant information at some point during a computation. For a URM U this includes the values of all variables and the *instruction counter*, which indicates which instruction has to be executed next, or whether the execution of the program is finished already. The action of U for a given input is then defined as a sequence of configurations.

The second approach (*denotational semantics*) proceeds by associating with every URM $U = (R, P)$ a mapping Φ_U, also denoted by Φ_P, if R is given by the context. Φ_P indicates how the execution of P effects the values of all the variables. It is defined by induction on the structure of P; for instance, if P is of the form $P_1; P_2$, then Φ_P is expressed in terms of Φ_{P_1} and Φ_{P_2}. After this has been accomplished, it is a trivial matter to define the *function computed by* P.

From the point of view of formal elegance the second approach is definitively superior. The first, on the other hand, fits in with the notion of programs as representations of algorithms or procedures in the sense of definition 1.1. Moreover, it shows URM's as a special case of machine as defined in section 4.2. In the further development of the theory, there are also some tasks which can be carried out more easily on the basis of denotational semantics (definition of the semantics of recursive procedures, proof of some theorems like 4.1), and others which are facilitated by operational semantics (handling of **goto**-statements, construction of the universal function in chapter 3).

We choose to apply operational semantics for our definitions. However, we will also define the mappings Φ_U on that basis, and prove that they satisfy the properties which would be used for their definition if denotational semantics were applied. This is the contents of theorem 1.1. By the way, the choice between the two approaches also has some implications for the definition of the syntax. If denotational semantics is taken as a basis, then it is natural to regard a complete **while**-loop (from **while** to the matching **end**) as a single statement, and to define the set of $\mathcal{WL}$-programs by rules like the following: every $\mathcal{WL}$-statement is a $\mathcal{WL}$-program; if P_1 and P_2 are $\mathcal{WL}$-programs, then so are $P_1; P_2$ and **while** $X > 0$ **do**; P_1; **end** etc. (see exercise 1.1 (iii)). This is how most textbooks proceed. However, under our approach, based on operational semantics, it is necessary to regard the line '**while** $X > 0$ **do**,' the corresponding **end**, and the lines in between as separate instructions, as it was done in definition 1.2. The reason is that the next instruction to be executed may be any instruction between **while** and **end**, see part (ii) of definition 1.4.

Notation: If ϕ is a function (with arbitrary domain and range), then $\phi\langle a/x\rangle$ is the function defined by

$$\phi\langle a/x\rangle(y) = \begin{cases} a & \text{if } y = x, \\ \phi(y) & \text{otherwise.} \end{cases}$$

Definition 1.4: (V) Let $U = (R, P)$ be a URM, $P = I_0; I_1; \ldots I_{l-1}$.

(i) A *configuration* (*instantaneous description*) $\mathcal{C}$ of U is a pair (C, V) such that
   (a) C is an integer, $0 \leq C \leq l$; C is called the *instruction counter* (of $\mathcal{C}$).
   (b) V is a total mapping from R to ω; $V(X)$ is the *value of the variable* X (in the configuration $\mathcal{C}$).

(ii) Let $\mathcal{C} = (C, V)$ and $\mathcal{C}' = (C', V')$ be configurations of U. Then $\mathcal{C}$ *yields* $\mathcal{C}'$ in one step (of a computation) ($\mathcal{C} \underset{U}{\longrightarrow} \mathcal{C}'$), if one of the following conditions holds:
   (a) Instruction I_C is of the form $X := X+1$, $C' = C+1$, and $V' = V\langle V(X)+1/X\rangle$.
   (b) I_C is of the form $X := X \dot{-} 1$, $C' = C + 1$, and $V' = V\langle V(X) \dot{-} 1/X\rangle$.

(c) I_C is of the form **while** $X > 0$ **do**, I_j is the corresponding **end**, that is, $j = r(C)$ in the notation of definition 1.2 (v), $V' = V$, and

$$C' = \begin{cases} C+1 & \text{if } V(X) > 0, \\ j+1 & \text{if } V(X) = 0. \end{cases}$$

(Thus a **while**-statement is executed by moving to the next statement or jumping after the corresponding **end**, depending on whether or not the loop has to be executed one more time.)

(d) I_C is an **end**-statement, I_k is the corresponding **while**-statement, $V' = V$, and $C' = k$. (Thus the execution of an **end**-statement consists in jumping to the corresponding **while**.)

Note that for every configuration $\mathcal{C}$ there exists at most one $\mathcal{C}'$ such that $\mathcal{C} \underset{U}{\longrightarrow} \mathcal{C}'$.

(iii) A *halting configuration* of U is a configuration $\mathcal{C} = (C, V)$ such that no configuration $\mathcal{C}'$ of U satisfies $\mathcal{C} \underset{U}{\longrightarrow} \mathcal{C}'$. (This is the case exactly if $C = l$.)

(iv) The *computation of* U *starting in the configuration* $\mathcal{C}$ (with *initial configuration* $\mathcal{C}$) is the (uniquely determined) sequence of configurations $\Gamma = (\mathcal{C}_0, \mathcal{C}_1, \ldots)$ such that $\mathcal{C}_0 = \mathcal{C}$, $\mathcal{C}_{i-1} \underset{U}{\longrightarrow} \mathcal{C}_i$ for $i = 1, 2, \ldots$, and such that Γ ends with the configuration $\mathcal{C}_t$ if and only if $\mathcal{C}_t$ is a halting configuration. In the latter case Γ *halts* (*terminates*) in the configuration $\mathcal{C}_t$, and $\mathcal{C}_t$ is the *final* configuration of Γ. Every transition from a configuration $\mathcal{C}_{i-1}$ to $\mathcal{C}_i$ is a *step* of the computation Γ.

(v) U computes a partial function $f\colon \omega^n \to \omega$ ($n \geq 0$) with respect to variables $X_1, \ldots, X_n$, $Y \in R$, iff the following holds for all $\vec{x}_n$: Let $\mathcal{C}_0$ be the configuration (C, V) of U such that $C = 0$, $V(X_i) = x_i$ for $1 \leq i \leq n$, and $V(X) = 0$ for all other $X \in R$. Then the computation Γ of U starting in the configuration $\mathcal{C}_0$ terminates if and only if $f(\vec{x}_n)\!\downarrow$, and if this is the case, then the final configuration of Γ is a configuration (C', V') such that $V'(Y) = f(\vec{x}_n)$. Γ is called the *computation of* U *for input* $\vec{x}_n$, if $X_1, \ldots, X_n$, Y are clear from the context. The *output variable* Y may be identical with one of the *input variables* X_i, but the input variables must be different from each other.

Usually a URM (R, P) will be given by specifying P alone; in this case it is understood that R is the set of all variables occurring in P. Also, we will sometimes say 'P computes f with respect to $X_1, \ldots$,' instead of 'U computes f with respect to X_1, $\ldots$.' This means that f is computed by some URM $U = (R, P)$, where R is any set of identifiers containing the variables of P and $X_1, \ldots$. Obviously, the *function computed by* P *with respect to* $X_1, \ldots$ is uniquely determined, since it does not depend on R.

The following notation applies not only to $\underset{U}{\longrightarrow}$, but to every binary relation:

Notation: Let $\longrightarrow$ be a binary relation on a set A. Then $\overset{+}{\longrightarrow}$ denotes the transitive closure, $\overset{*}{\longrightarrow}$ the reflexive and transitive closure of $\longrightarrow$. Hence $a \overset{+}{\longrightarrow} b$ holds iff there exists a sequence of elements $a_0 = a$, $a_1, \ldots, a_m = b$ ($m \geq 1$) such that $a_{i-1} \longrightarrow a_i$ for $1 \leq i \leq m$, and the criterion for $a \overset{*}{\longrightarrow} b$ is similar, except that m may be 0, in which case $a = b$.

Thus the computation Γ starting in a configuration $\mathcal{C}$ terminates in the configuration $\mathcal{C}'$ iff $\mathcal{C} \xrightarrow[U]{*} \mathcal{C}'$ and $\mathcal{C}'$ is a halting configuration, and Γ does not terminate iff no such $\mathcal{C}'$ exists. This notation was not needed in definition 1.4, but it will be useful later, see, for instance, definition 4.4.

Exercise 1.2: [0] What function is computed by the empty $\mathcal{WL}$-program with respect to $X_1, \ldots, X_n, Y$ (i) if Y is the same as X_i, (ii) if $Y \notin \{X_1, \ldots, X_n\}$?

Exercise 1.3: [3] It was argued informally above that the URM U from example 1.1 computes $\lambda xy.x + y$ with respect to X, Y and Y. Now give a rigorous mathematical proof based on definition 1.4.

In the following three exercises you are not required to prove rigorously that your $\mathcal{WL}$-programs do the job. Just proceed as usual when you write a program (except that you have to observe the syntax of $\mathcal{WL}$).

Exercise 1.4: Write *special* $\mathcal{WL}$-programs which compute the following functions with respect to the given variables. In each case they are just as easy to construct as arbitrary $\mathcal{WL}$-programs for the given functions would be.

(i) [1] $\lambda \vec{x}_n.c$ (c a constant), with respect to $X_1, \ldots, X_n, Y$.
(ii) [0] $\lambda xy.x \dot{-} y$, with respect to X, Y, X.
(iii) [3] $\lambda xy.xy$, with respect to X, Y, Z.
(iv) [2] $\lambda x. \operatorname{sig} x$, with respect to X, Y, where

$$\operatorname{sig} x = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

Exercise 1.5: Write (arbitrary) $\mathcal{WL}$-programs which compute the following functions with respect to the given variables. (This time no special $\mathcal{WL}$-programs exist, since special $\mathcal{WL}$-programs always compute *total* functions.)

(i) [0] For every n the nowhere defined function (P) $\lambda \vec{x}_n.\uparrow$, with respect to $X_1, \ldots, X_n$, Y. (This notation is not really an instance of the lambda-notation, since $\uparrow$ is not a 'value;' however, its meaning is obvious, and we will continue to use it.)
(ii) [4] the partial function f such that

$$f(x, y) = \begin{cases} x - y & \text{if } x \geq y, \\ \uparrow & \text{if } x < y, \end{cases}$$

with respect to X, Y, X.

Exercise 1.6: Let $f(x) = \lfloor x/2 \rfloor$; this is an example where a special $\mathcal{WL}$-program is harder to construct than a non-special one.

(i) [2] Write an (arbitrary) $\mathcal{WL}$-program which computes f with respect to X and Y.
(ii) [4] Write a special $\mathcal{WL}$-program which computes f with respect to X and Y.

Next we define the mapping Φ_U for any given URM U, as promised above. Let $U = (R, P)$ be a URM, and let $\mathcal{V}$ be the set of all functions from R to ω (systems of values of the variables). Φ_U characterizes the changes of values effected by U. Formally, Φ_U is defined as follows. Given $V \in \mathcal{V}$, let $\mathcal{C}$ be the configuration $(0, V)$ of U, and let Γ be the computation of U starting in the configuration $\mathcal{C}$. If Γ terminates in the configuration $\mathcal{C}' = (C', V')$, then $\Phi_U(V) = V'$. If Γ does not terminate, then $\Phi_U(V)$ is undefined.

Intuitively, the assertions of the following theorem are quite obvious (at least for the most part). The proof serves as an example as to how such 'obvious' facts can be shown.

Theorem 1.1: Let $U = (R, P)$ be a URM. In what follows write $\Phi_{P'}$ instead of $\Phi_{U'}$, for any URM $U' = (R, P')$ with the same set of registers as U. In (iii), (iv) it is assumed that P_1, P_2 are $\mathcal{WL}$-programs.

(i) If P is empty, then $\Phi_P(V) = V$ for all V.

(ii) If P consists of the single assignment statement $X := X + 1$ or $X := X \dot{-} 1$, then $\Phi_P(V) = V\langle V(X) + 1/X\rangle$ or $V\langle V(X) \dot{-} 1/X\rangle$ respectively.

(iii) If P is of the form P_1; P_2, then $\Phi_P(V) = \Phi_{P_2}\Phi_{P_1}(V)$.

(iv) If P is of the form **while** $X > 0$ **do**; P_1; **end**, then

$$\Phi_P(V) = \begin{cases} V & \text{if } V(X) = 0, \\ \Phi_P\Phi_{P_1}(V) & \text{if } V(X) > 0. \end{cases} \tag{1}$$

Moreover, if Φ' is any other mapping from $\mathcal{V}$ to $\mathcal{V}$ such that (1) is satisfied with Φ' in place of Φ_P, then $\Phi_P \subseteq \Phi'$.

(Note that one of the four cases always applies by exercise 1.1 (iii).)

Proof: Let $V \in \mathcal{V}$, $\mathcal{C}_0 = (0, V)$, and let $\Gamma = (\mathcal{C}_0, \mathcal{C}_1, \ldots)$ be the computation of U starting in configuration $\mathcal{C}_0$, as in the definition of Φ_P.

(i) If P is empty, then $\mathcal{C}_0$ is already a halting configuration and hence the final configuration of Γ. The assertion follows, see exercise 1.2.

(ii) If P consists of the single assignment $X := X + 1$ or $X := X \dot{-} 1$, then $\mathcal{C}_0 = (0, V)$, $\mathcal{C}_1 = \big(1, V\langle V(X) + 1/X\rangle\big)$ or $\big(1, V\langle V(X) \dot{-} 1/X\rangle\big)$ respectively, and $\mathcal{C}_1$ is a halting configuration. The assertion follows.

Now let $P = (I_0, \ldots, I_{l-1})$ be of the form given in (iii) or (iv). Before proceeding further, we note:

Lemma : Let $P' = (I_{l_0}, \ldots, I_{l_1-1})$ be a segment of P which also forms a $\mathcal{WL}$-program. Let $\mathcal{C}'_0 = (C_0, V_0)$ be a configuration of $U' = (R, P')$, $\Gamma' = (\mathcal{C}'_0, \mathcal{C}'_1, \ldots)$ the computation of U' starting in $\mathcal{C}'_0$, $\mathcal{C}'_i = (C_i, V_i)$, and let $\mathcal{C}_i = (C_i + l_0, V_i)$. Then the computation Γ of U starting in configuration $\mathcal{C}_0$ begins with $(\mathcal{C}_0, \mathcal{C}_1, \ldots)$ (and proceeds further if Γ' terminates and $l_1 < l$). Thus if $(C, V) \xrightarrow[U']{*} (\tilde{C}, \tilde{V})$, then $(C + l_0, V) \xrightarrow[U]{*} (\tilde{C} + l_0, \tilde{V})$.

The proof of this lemma is easy and left for exercise 1.7. We just remark that it uses the last assertion in exercise 1.1 (iv).

We now turn to the proof of (iii) and (iv). Let $U_i = (R, P_i)$ for $i = 1, 2$.

(iii) Let $P_1 = (I_0, \ldots, I_{l_0-1})$, $P_2 = (I_{l_0}, \ldots, I_{l-1})$, and let Γ_1 be the computation of U_1 starting in configuration $(0, V)$. If Γ_1 terminates, then its final configuration is $\mathcal{C}' = \big(l_0, \Phi_{P_1}(V)\big)$; in this case let Γ_2 be the computation of U_2 starting in configuration $\mathcal{C}^{(1)} = \big(0, \Phi_{P_1}(V)\big)$. Assuming that Γ_2 terminates as well, its final configuration is $\mathcal{C}'' = \big(l - l_0, \Phi_{P_2}\Phi_{P_1}(V)\big)$. Applying the lemma twice we get

$$\mathcal{C}_0 \xrightarrow[U]{*} \big(l_0, \Phi_{P_1}(V)\big) \xrightarrow[U]{*} \big(l, \Phi_{P_2}\Phi_{P_1}(V)\big),$$

hence $\Phi_P(V) = \Phi_{P_2}\Phi_{P_1}(V)$, as claimed. If on the other hand one of the computations Γ_1 or Γ_2 does not terminate, then neither does Γ, hence $\Phi_P(V)$ and $\Phi_{P_2}\Phi_{P_1}(V)$ are both undefined, and the assertion follows again.

(iv) Note first that the initial **while**-statement matches with the final **end** (I_{l-1}); this follows readily by exercise 1.1 (iv). If $V(X) = 0$, then $(0, V) \underset{U}{\longrightarrow} (l, V)$ according to definition 1.4 (ii) (c), hence Γ terminates in (l, V), and $\Phi_P(V) = V$, as claimed.

Now assume $V(X) > 0$. Then the first step of Γ transforms $(0, V)$ to $(1, V)$. Let Γ_1 be the computation of $U_1 = (R, P_1)$ starting in $\mathcal{C}^{(1)} = (0, V)$. If Γ_1 terminates, then its final configuration is $\big(l-2, \Phi_{P_1}(V)\big)$. Using the lemma and definition 1.4 (ii) (d), we have

$$(0, V) \underset{U}{\longrightarrow} (1, V) \overset{*}{\underset{U}{\longrightarrow}} \big(l-1, \Phi_{P_1}(V)\big) \underset{U}{\longrightarrow} \big(0, \Phi_{P_1}(V)\big).$$

Let Γ' be the computation of U starting in $\big(0, \Phi_{P_1}(V)\big)$. If Γ' also terminates, then its final configuration is $\big(l, \Phi_P\Phi_{P_1}(V)\big)$, we have $(0, V) \overset{*}{\underset{U}{\longrightarrow}} \big(l, \Phi_P\Phi_{P_1}(V)\big)$, and the assertion follows. If on the other hand, either Γ_1 or Γ' does not terminate, then neither does Γ, and $\Phi_P(V)$ and $\Phi_P\Phi_{P_1}(V)$ are both undefined, again in accordance with (1).

Finally, let Φ' be some other mapping from $\mathcal{V}$ to $\mathcal{V}$ satisfying (1). Let $V \in \mathcal{V}$ be such that $\Phi_P(V)\!\downarrow$; we have to show that $\Phi'(V) = \Phi_P(V)$. This is done by induction on the length of Γ, where Γ is once again the computation of U starting in $(0, V)$. Hence we may assume that the assertion holds for every $V' \in \mathcal{V}$ such that the computation of U starting in $(0, V')$ is shorter than Γ.

If $V(X) = 0$, then $\Phi_P(V) = \Phi'(V) = V$ since Φ_P and Φ' satisfy (1). If $V(X) > 0$, then let Γ' be the computation of U starting in $\big(0, \Phi_{P_1}(V)\big)$, as above. It was shown that Γ' is a segment of Γ, so Γ' is shorter than Γ. Hence $\Phi'\Phi_{P_1}(V) = \Phi_P\Phi_{P_1}(V)$ by induction hypothesis. Once again it follows that $\Phi'(V) = \Phi_P(V)$, since Φ_P and Φ' satisfy (1). This completes the proof. □

Exercise 1.7: [2] Prove the lemma in the proof of theorem 1.1.

Now we are in a position to show how the function f computed by U (or P) with respect to $X_1, \ldots, X_n, Y$ can be defined by denotational semantics. Note first that theorem 1.1 provides a complete characterization of the mappings Φ_U for all URM's U; this would not be the case without the last statement in (iv). Thus an alternative definition of Φ_U is obtained. Given $(x_1, \ldots, x_n)$, let $V \in \mathcal{V}$ be defined by

$$V(X) = \begin{cases} x_i & \text{if } X = X_i, \\ 0 & \text{if } X \notin \{X_1, \ldots, X_n\}. \end{cases}$$

Then $f(x_1, \ldots, x_n) = \Phi_U(V)(Y)$.

Example 1.3: Let P be the $\mathcal{WL}$-program of example 1.1, $R = \{X, Y\}$, $U = (R, P)$. Also, let P_1–P_3 be segments of P such that P_1 consists only of line 1, P_2 of line 2, P_3 of lines 1 and 2 of P. Let $V\colon R \to \omega$ be such that $V(X) = x$, $V(Y) = y$; we write simply $V = (x, y)$. Then by theorem 1.1 we have

$$\begin{aligned} \Phi_{P_1}(V) &= (x \dot{-} 1, y), \\ \Phi_{P_2}(V) &= (x, y + 1), \\ \Phi_{P_3}(V) &= (x \dot{-} 1, y + 1). \end{aligned}$$

We now show by induction on x that

$$\Phi_P(V) = (0, x + y), \tag{2}$$

using part (iv) of theorem 1.1. If $x = 0$, then $\Phi_P(V) = V$, which agrees with (2). If $x > 0$, then by induction hypothesis we have

$$\Phi_P(V) = \Phi_P\Phi_{P_3}(V) = \Phi_P\big((x - 1, y + 1)\big) = (0, x - 1 + y + 1) = (0, x + y),$$

and we are done.

Exercise 1.8: [3] For the URM constructed in exercise 1.4 (iv), prove formally that it computes $\lambda x.\operatorname{sig} x$ with respect to X and Y, using the method of example 1.3.

Exercise 1.9: [2] Let $U = (R, P)$, $U' = (R', P)$ be URM's with the same program, where $R \subseteq R'$. Also, let $V\colon R \to \omega$, $V'\colon R' \to \omega$ such that $V(X) = V'(X)$ for $X \in R$, $W = \Phi_U(V)$, $W' = \Phi_{U'}(V')$. Show that

$$W'(X) = \begin{cases} W(X) & \text{for } X \in R, \\ V'(X) & \text{for } X \in R' - R. \end{cases}$$

This means that the function computed by a URM U with respect to a given tuple of variables remains unchanged if additional registers are added, and that the computation by U does not alter the values of variables not occurring in P. This is another case where a formal proof for an intuitively obvious assertion is asked for.

Exercise 1.10: (i) [4] Describe the set of all functions which are computed by programs consisting only of $\mathcal{WL}$-assignments, no **while**- and **end**-statements. (The answer is slightly trickier than it might appear at first glance.)

(ii) [4] Describe the set of all functions which are computed by $\mathcal{WL}$-programs without assignments of the form $X := X \dot{-} 1$.

1.3. The fundamental notions of recursive function theory.

As announced at the beginning of section 1.2, we now use the language $\mathcal{WL}$ to introduce fundamental classes of functions:

Definition 1.5: (E) A function f is *primitive recursive*, iff it is computed by some special $\mathcal{WL}$-program; f is *partial recursive*, iff it is computed by an (arbitrary) $\mathcal{WL}$-program (with respect to some tuple of variables in both cases); finally, f is *recursive*, iff f is partial recursive and total. The set of all primitive recursive, recursive and the set of all partial recursive functions are denoted by $\mathcal{PR}$, $\mathcal{R}$ and $\mathcal{P}$, respectively.

The motivation for the consideration of $\mathcal{P}$ and $\mathcal{R}$ is clear in view of the remarks from section 1.2. The partial recursive functions are regarded as a mathematical equivalent for the 'algorithmically computable' functions, and, naturally, the total ones among them are also of interest. For $\mathcal{PR}$ the motivation is less obvious. It is just a subclass of $\mathcal{R}$ for which a uniform representation (by special $\mathcal{WL}$-programs) exists. This is also true for other subsets of $\mathcal{R}$, though $\mathcal{PR}$ has tradionally been dealt with in textbooks. But classical recursive function theory can be developed without dealing with primitive recursive functions.

All this does not mean, however, that the class $\mathcal{PR}$ is not of interest. Roughly speaking, a function $f: \omega^n \to \omega$ is primitive recursive iff f has a representation which makes it immediately obvious that f is computable and total. This characterization holds with respect to several formalism for defining the class of computable functions.

For instance, the concept of computability may be based on some high-level Algol-like programming language. All control structures may be divided into two groups: those which do not allow to introduce infinite loops (type A), and those which allow it (type B). Type A includes conditional statements, **case**-statements, **for**-loops and non-recursive procedures and functions, type B **while**- and **repeat**-**until**-loops, **goto**-statements and recursive procedures and functions. (In these lecture notes we apply the Pascal-form of **for**-loops, namely **for** $I := e_1$ **to** e_2, where e_1 and e_2 are evaluated in advance. Hence the number of executions is also predetermined. In the current context, **while**-loops are of course not restricted as in definition 1.2 (iii). In general, when we speak of a **while**-loop or **while**-statement I (without further specification), then no restriction is implied unless I belongs to a $\mathcal{WL}$-program, as it was the case so far.) The primitive functions can be characterized as those functions $f: \omega^n \to \omega$ which are computed by programs containing only control structures of type A, see theorem 2.1 (ii). By contrast, each control structure of type B allows to compute non-total functions, as well as recursive functions which are not primitive recursive. If a program P containing control structures of type B nevertheless computes a primitive recursive function, then P can be transformed into an equivalent program P', where all control structures are of type A. Such a transformation is not always practical, though, and P' may be more complicated and less readable than P.

Another equivalent characterization of $\mathcal{PR}$ will be given in theorem 4.1. $\mathcal{PR}$ is also relevant when recursive functions are classified according to their complexity; for this purpose the distinction between primitive recursive and other recursive functions may serve as a first step. Finally, it should be noted that virtually all computable functions encountered in practice are in fact primitive recursive.

So far a number of proofs of 'obvious' assertions about the behavior of certain programs were provided or asked for in the exercises. They served primarily as a practice for handling the formalism of definitions 1.2–1.4. The following will be the last proof of this kind in these lecture notes. Similar assertions will no longer be shown if they are intuitively clear.

Theorem 1.2: Every primitive recursive function is total.

Proof: It suffices to show that Φ_U is total for every URM $U = (R, P)$ with P special; this implies that the function computed by U (with respect to any tuple of variables) is also total. We apply induction on the length of P, hence assume that the assertion holds for every URM $U' = (R', P')$ with P' special and shorter than P.

If P is empty or consists of a single assignment statement, then trivially Φ_U is total (theorem 1.1 (i), (ii)). If P is of the form $P_1; P_2$, where P_1 and P_2 are also $\mathcal{WL}$-programs, then P_1 and P_2 are also special by exercise 1.1 (v), hence Φ_{P_1} and Φ_{P_2} are total by induction hypothesis. (As in theorem 1.1 Φ_{P_i} means Φ_{U_i}, where $U_i = (R, P_i)$.) Then $\Phi_U = \Phi_P = \Phi_{P_1}\Phi_{P_2}$ by theorem 1.1 (iii), so once again Φ_U is total.

By exercise 1.1 (iii), (v) it remains to consider the case where P is of the form

$$\textbf{while } X > 0 \textbf{ do; } X := X \dot{-} 1;\ P_1;\ \textbf{end},$$

P_1 is special and contains no assignments to the variable X. Actually, X does not occur in P_1 at all, for if P_1 contains the statement **while** $X > 0$ **do**, then P_1 would also contain the assignment $X := X \dot{-} 1$, since P_1 is special. Let P_2 be the program $X := X \dot{-} 1; P_1$, and let $\mathcal{V}$ be the set of all mappings from R to ω, as in theorem 1.1. We have to show that $\Phi_P(V)\downarrow$ for all $V \in \mathcal{V}$; this is done by induction on $V(X)$.

If $V(X) = 0$, then $\Phi_P(V) = V$ by theorem 1.1 (iv), and we are done. Let $V(X) > 0$, and assume that the assertion holds for all $V' \in \mathcal{V}$ with $V'(X) = V(X) - 1$. Using again theorem 1.1 (iv), we have $\Phi_P(V) = \Phi_P \Phi_{P_2}(V)$. Here $V' = \Phi_{P_2}(V)\downarrow$ by the induction hypothesis for the outer induction (P_2 in place of P). Also, $V' = \Phi_{P_1}(V\langle V(X) \dot{-} 1/X\rangle)$, hence $V'(X) = V(X) - 1$, since X does not occur in P_1. (Formally, this step of the argument uses exercise 1.9 for the URM's $(R - \{X\}, P_1)$ and (R, P_1).) Applying the induction hypothesis for the inner induction (V' in place of V), it follows that $\Phi_P(V) = \Phi_P(V')\downarrow$, completing the proof. □

A set $A \subseteq \omega^n$ is called *decidable*, iff there exists some algorithm, as characterized by $\mathcal{WL}$-programs, which decides for any given $\vec{x}_n$ whether or not $\vec{x}_n \in A$. Since only functions from ω^n to ω are considered, outputs 1 and 0 are interpreted as 'yes' and 'no.' The formal definition is as follows:

Definition 1.6: Let $A \subseteq \omega^n$.

(i) The *characteristic function* of A is defined by

$$c_A(\vec{x}_n) = \begin{cases} 1 & \text{if } \vec{x}_n \in A, \\ 0 & \text{if } \vec{x}_n \notin A. \end{cases}$$

If P is the predicate $\lambda\vec{x}_n.\vec{x}_n \in A$, then the *characteristic function of* P is again c_A, and it is also denoted by c_P.

(ii) A is *primitive recursive*, iff c_A is primitive recursive; A is *recursive* (*decidable*), iff c_A is recursive. (These properties also apply to the corresponding predicates, see the conventions from the introduction.)

Example 1.4: (i) The sets $\emptyset$ and ω^n, which correspond to the predicates $\lambda\vec{x}_n.$**false** and $\lambda\vec{x}_n.$**true**, are primitive recursive. Their characteristic functions are $\lambda\vec{x}_n.0$ and $\lambda\vec{x}_n.1$. $\lambda\vec{x}_n.0$ is computed by the empty program with respect to variables $X_1, \ldots, X_n$ and $Y \notin \{X_1, \ldots, X_n\}$ (see exercise 1.2), and $\lambda\vec{x}_n.1$ is computed by the program consisting of the single statement $Y := Y + 1$ with respect to the same variables.

We note in passing that the argument from the preceding paragraph also applies for $n = 0$. In this case $\emptyset$ and $\omega^0 = \{()\}$ are the *only* subsets of ω^n.

(ii) The predicate $P = \lambda x.x > 0$ is primitive recursive. This follows from exercise 1.4 (iv), since sig is the characteristic function of P.

Exercise 1.11: Show that the following predicates are primitive recursive:

(i) [1] $\lambda x.x < c$, where c is a constant.

(ii) [2] $\lambda xy.x < y$.

(iii) [2] $\lambda xy.x = y$.

Notation: In general, if $\mathcal{F} \subseteq \mathcal{NF}$, then $\mathcal{F}_*$ denotes the set of all sets (or predicates) $A \in \mathcal{NS}$ such that $c_A \in \mathcal{F}$. Accordingly, the class of all primitive recursive and the class of all recursive sets (of predicates) are denoted by $\mathcal{PR}_*$ and $\mathcal{R}_*$ respectively. Also, $\mathcal{F}^{(n)}$ denotes the set of all functions in $\mathcal{F}$ with n arguments, and similarly, if $\mathcal{S} \subseteq \mathcal{NS}$, then $\mathcal{S}^{(n)}$ denotes the set of all sets in $\mathcal{S}$ which are subsets of ω^n. For instance, $\mathcal{R}^{(n)}$ denotes the set of all recursive functions of n arguments, $\mathcal{PR}_*^{(n)}$ denotes the set of all primitive recursive subsets of ω^n etc.

The last of the fundamental concepts of computability theory to be introduced is that of *recursively enumerable* (*r.e.*) set. Intuitively, there are two different ways to view r.e. sets, which turn out to be equivalent. For one thing, a set A is r.e., iff it can be *generated* (*enumerated*) by some procedure. Such a procedure yields an output from time to time, and A is the set of all numbers which are eventually given as output.

Alternatively, the set A is r.e., iff it is *accepted* by some procedure P. This means that for input x P behaves as follows: If $x \in A$, then this fact is recognized after a finite number of steps, and an output like 'yes' is given. If, on the other hand, $x \notin A$, then P may give the answer 'no', but it may also happen that P gives no answer at all and runs forever.

Example 1.5: A number n is called a *Fermat-exponent*, if there exist no integers x, y, $z > 0$ such that $x^n + y^n = z^n$, otherwise n is a *non-Fermat-exponent*. It was shown only 1994 by A. Wiles and R. Taylor that all $n \geq 3$ are Fermat-exponents; prior to that it was also an open question whether or not the set of all Fermat-exponents is recursive.

Let A be the set of all non-Fermat-exponents. It is easy to design a procedure which systematically checks all quadruples (x, y, z, n) with x, y, $z > 0$ and outputs n whenever $x^n + y^n = z^n$; such a procedure generates A. Also, given n, we may systematically check for all x, y, $z > 0$ if $x^n + y^n = z^n$ holds, and output 'yes' if this is the case for some x, y, z; this yields a procedure which accepts A. Hence we may argue informally that A is r.e, without using the Wiles-Taylor result. A rigorous proof is provided in exercise 2.10 (ii).

If $\operatorname{dom} f = A$, then a procedure for computing f may also be regarded as a procedure for accepting A, since we may interpret every output as a code for 'yes.' Therefore the following formal definition agrees with the intuition given above.

Definition 1.7: (E) A set $A \subseteq \omega^n$ is *recursively enumerable* (*r.e.*), iff it is the domain of some partial recursive function. The class of all r.e. sets is denoted by $\mathcal{E}$.

The following theorem indicates how the various classes of functions and sets introduced in this section are related. It is mainly a summary, since most assertions are already implicit in previous results.

Theorem 1.3: (i) $\mathcal{PR} \subseteq \mathcal{R} \subsetneq \mathcal{P}$.
(ii) $\mathcal{PR}_* \subseteq \mathcal{R}_* \subseteq \mathcal{E}$.

Proof: (i) The relation $\mathcal{PR} \subseteq \mathcal{R} \subseteq \mathcal{P}$ follows immediately from definition 1.5 and theorem 1.2. An example of a partial recursive function which is not total is $\lambda\vec{x}_n.\uparrow$ for any $n \in \omega$ (exercise 1.5 (i)); hence the second subset relation is proper.

(ii) The relation $\mathcal{PR}_* \subseteq \mathcal{R}_*$ follows from $\mathcal{PR} \subseteq \mathcal{R}$, so it remains to show that every recursive set A is r.e. Let P be a $\mathcal{WL}$-program which computes c_A with respect to X_1, $\ldots$, X_n, Y. Consider the following program P':

P;
$Z := Z + 1$;
while $Y > 0$ **do**;
$Y := Y \dot{-} 1$;
$Z := Z \dot{-} 1$;
end;
while $Z > 0$ **do**;
end

The execution of P' for input $\vec{x}_n$ proceeds as follows. After the assignment $Z := Z + 1$, Z has value 1. (Recall that every variable except for the input variables is initialized to 0.) If $\vec{x}_n \in A$, then P assigns the value 1 to Y, hence the first **while**-loop after P changes the value of Z to 0, the second **while**-loop does not involve any action, and the computation terminates. If $\vec{x}_n \notin A$, then the first **while**-loop leaves the value of Z at 1, and the second (with empty body) is executed an infinite number of times. (The computation jumps between **while** $Z > 0$ **do** and **end** forever.) It follows that the function computed by P' with respect to $X_1, \ldots, X_n, Y$ has domain A; the assertion follows. □

Actually, every subset relation in theorem 1.3 is proper: there exist recursive functions and predicates which are not primitive recursive, and r.e. sets which are not recursive. However, these facts and other interesting assertions about r.e. sets would be rather difficult to prove at the present stage. They will be shown in chapter 3, when additional tools are available.

2. Constructions of primitive recursive and partial recursive functions.

The purpose of this chapter is to give some indication as to the size of the classes $\mathcal{PR}$, $\mathcal{R}$ and $\mathcal{P}$. It will be seen that, roughly speaking, $\mathcal{PR}$ includes all 'practically' computable total functions, virtually all functions computed by programs written in real life, and many more. Nevertheless, it is possible to construct programs which compute total functions not in $\mathcal{PR}$; this is just a restatement of the assertion $\mathcal{PR} \subsetneq \mathcal{R}$, which was made after the proof of theorem 1.3. The size of $\mathcal{R}$ and $\mathcal{P}$ is indicated simply by the fact that ChT has not been refuted; in other words, $\mathcal{P}$ contains all partial functions computed by known classes of procedures, and $\mathcal{R}$ contains all total functions of this kind.

2.1. An extended programming language.

As noted, we wish to prove that many classes of functions are partial recursive or even primitive recursive, specifically:

(a) functions computed by programs of high-level languages, and

(b) functions obtained by certain mathematical definition schemes.

In this section we go some distance towards the first goal. It would be possible to state in general terms that conditional statements, **while**-loops of a general form etc. can be simulated in $\mathcal{WL}$. However, syntax and semantics of some constructs vary from language to language, hence we choose to introduce a new programming language $\mathcal{AL}$, in order to avoid any ambiguities that might arise. It will be shown (theorem 2.1) that many constructs of $\mathcal{AL}$ can be simultated in $\mathcal{WL}$; others will be dealt with later. Many constructs of real programming languages occur or have analogues in $\mathcal{AL}$, and the results of this section apply to them as well.

Actually, the language to be introduced is $\mathcal{AL}(\mathcal{F})$, where $\mathcal{F}$ is a list of data types, functions and predicates to be used. The motivation is as follows: by theorems 2.1, 2.13 and 2.15 $\mathcal{AL}(\mathcal{F})$-programs may be used to deduce that certain specific functions are partial recursive. The condition is that the functions (predicates) in $\mathcal{F}$ are partial recursive (recursive). As we proceed, the set of types, functions and predicates that may be included in $\mathcal{F}$ increases, and it would be inappropriate to fix it once and forever.

We now describe syntax and semantics of $\mathcal{AL}(\mathcal{F})$. This will be done in a somewhat sketchy fashion, as opposed to the rigorous definitions for $\mathcal{WL}$.

Define *$\mathcal{F}$-terms* (*$\mathcal{F}$-conditions*) to be expressions E of the form $f(X_1, \ldots, X_n)$ $Q(X_1, \ldots, X_n)$), where f (Q) is a function (predicate) in $\mathcal{F}$, and X_1, …, X_n are variables. In both cases E may also have a different syntactical form, provided that $\lambda X_1 \ldots X_n.E$ is a function or predicate respectively in $\mathcal{F}$. For instance, infix notation may be used, so $X + Z$ is a $\mathcal{F}$-term if $\lambda xy.x + y \in \mathcal{F}$, and $X < Y$ is a $\mathcal{F}$-condition, if $\lambda xy.x < y \in \mathcal{F}$. A variable alone is a $\mathcal{F}$-term iff $\mathcal{F}$ contains $\lambda x.x$. See remark 2.1.

Next we provide a list of constructs that are allowed in $\mathcal{AL}(\mathcal{F})$-programs. The constructs listed under (ii)–(x) and (xii), as well as procedure calls (see (xi)) are called *$\mathcal{AL}$-statements*. This syntactical category is defined *recursively*, since it is in turn used in the definition of conditionals, generalized **while**-statements etc. Thus **then**- and **else**-part of a conditional may be a **for**-statement, whose body may be a compound statement etc. In what follows the symbols E, Q and S (sometimes with subscripts) represent $\mathcal{F}$-terms, $\mathcal{F}$-conditions and $\mathcal{AL}$-statements respectively.

(i) Variable declarations: the declaration $A{:}\,T$ indicates that A has data type $T \in \mathcal{F}$.

(ii) Empty statements.

(iii) Compound statements of the form **begin** S_1; ...; S_k **end**. (The number k may be ≤ 1, though in this case the delimiters **begin**, **end** are usually redundant. Exception: **begin** ... **end** is needed for the body of a procedure declaration, even if it contains only one statement (or even if it is empty).

(iv) Assignment statements of the form $Y := E$.

(v) Conditional statements of the form **if** Q **then** S_1 **else** S_2 or **if** Q **then** S.

(vi) **case**-statements, which have the form

case I **of**
  E_1: S_1;
  ⋮
  E_n: S_n;
  else S

where $E_1, \ldots, E_n$ are of the same data type as I. The effect is that S_i is executed if i is minimal such that I has the same value as E_i, and S is executed if no such i exists. S may be empty, of course, but the **else**-part cannot be omitted as in conditional statements, since it also serves to delimit the **case**-statement. (In real programming languages the expressions E_i are usually required to be distinct constants.)

(vii) (Generalized) **while**-statements of the form **while** Q **do** S. A statement of the form **while** $X > 0$ **do begin** $X := X \dot{-} 1$; S' **end** is called a *special* **while**-*statement*, iff S' contains no assignments to X and no procedure calls with X as an actual variable parameter.

(viii) **for**-statements: **for** $I := E_1$ **to** E_2 **do** S. I is the *control variable* of this statement. In (vii) and (viii) S is called the *body* of the **while**- or **for**-statement.

(ix) Labelled statements of the form L: S, where L is an identifier and S is an unlabelled $\mathcal{AL}$-statement.

(x) **goto**-statements: **goto** L effects an unconditional jump to the statement labelled L. Of course, conditional jumps are also possible by combining **goto** with **if** ... **then**.

(xi) Declarations and calls of *procedures*.

(xii) enum- and def-statements.

Two of the features listed will be dealt with only in later sections, though they were included in our list for the sake of completeness. Data types other than ω will be considered in sections 2.3 and 2.4, enum- and def-statements in section 3.3, when we consider programs for generating r.e. sets or partial recursive functions. For the time being, '$\mathcal{AL}$-program' really means '$\mathcal{AL}$-program without type declarations, enum- and def-statements.' In particular, theorems 2.1 and 2.3 are understood in this sense; there is no need to repeat the restrictions from the current paragraph, though others have to be made. The default type for non-declared variables is ω; this type is always present and need not be included in $\mathcal{F}$.

From a theoretical point of view, **case**-statements are of little importance; in fact, it is obvious that they can be replaced easily by nested conditionals. However, they will come in handy in the next to last step of the proof of theorems 2.1 and 2.2, and also in the proof of theorem 2.15. Other constructs were included in the list because they (or their analogues) form a very important part of programming languages in general.

We now turn to some details of the syntax of $\mathcal{AL}$ which remain to be specified.

Procedure declarations preceed the rest of the program (main program). They have the following form:

> **procedure** $Pr(X_1, \ldots, X_k;$ **var** $Y_1, \ldots, Y_l)$;
> **local** $Z_1, \ldots, Z_j$;
> **procedure** $Pr_1(\ldots)$;
> $\vdots$
> **procedure** $Pr_r(\ldots)$;
> $\vdots$
> **begin**
> S
> **end**

Here X_1, …, X_k are the formal *value parameters*, Y_1, …, Y_l the formal *variable parameters*; Z_1, …, Z_j are declared as *local* variables, and Pr_1, …, Pr_r as *local procedures*. (Of course, none of these must necessarily be present.) The identifiers X_1, …, X_k, Y_1, …, Y_l, Z_1, …, Z_j must be different from each other and also from identifiers used in other procedures, see below. The declarations of the local procedures have the same form as that of Pr. The portion **begin** S **end** is a compound $\mathcal{AL}$-statement; it is called the *body* of the procedure declaration.

A *call* of this procedure is of the form $Pr(E_1, \ldots, E_k, U_1, \ldots, U_l)$; here the *actual value parameters* E_1, …, E_k are $\mathcal{F}$-terms, and the *actual variable parameters* U_1, …, U_j are identifiers (variables). The actual parameters need not all be different, and the U_i's may also occur within the E_i's.

For simplicity we stipulate that no identifier X may be used for more than one purpose. This has the following consequences:

(a) An identifier X may only be used as one of the following: a label, a variable, or a procedure identifier.

(b) All statement labels of P are different, so there is no need to declare labels locally.

(c) If X is used as a variable identifier, then there is at most one procedure Pr such that X is a formal parameter or local variable of Pr; we say that X *belongs to* Pr. Also, X may be undeclared, in which case X is a *global variable* and belongs to P. If instead X is used as a procedure identifier, then the situation is similar, but of course X must be declared in every case.

Of course, this restriction on the use of identifiers is of no theoretical importance, since labels, local variables etc. may be renamed if necessary.

When we speak of *the procedures of* P, we mean P itself, along with all procedures declared in P or in some other procedure of P. If Pr is a procedure of P other than P, then the *parent procedure* of Pr is the procedure in which Pr is declared. The *ancestors* of Pr are Pr itself, the parent of Pr, the parent of the parent of Pr etc. If Pr' is an ancestor of Pr, then Pr is a *descendant* of Pr'. A variable or procedure X may be used in Pr iff it belongs to some ancestor of Pr. This includes the case where X is a procedure identifier and Pr and X have the same parent; now, in order to use X in Pr it is not required that the declaration of X preceeds that of Pr.

There may be jumps out of, but not into the bodies of **while**- and **for**-loops, procedure declarations and the **then**- and **else**-parts of conditional statements. The body of a **for**-loop with control variable I must not contain any assignments to I, nor any procedure calls with I as a variable parameter.

This completes the description of the syntax of $\mathcal{AL}(\mathcal{F})$. An *A-URM* is a pair (R, P), where P is an $\mathcal{AL}(\mathcal{F})$-program for some $\mathcal{F}$, and R is a set of registers containing all *global* variables of P.

Remark 2.1: According to our definition $\mathcal{F}$-terms and $\mathcal{F}$-conditions are of a rather restricted form. If an $\mathcal{AL}(\mathcal{F})$-program contains a more complicated expression E like $f\big(g(X,Y), h(X,Z)\big)$, then $\mathcal{F}$ must contain $\lambda XYZ.E$, not just f, g and h. In order to show that such programs always compute primitive recursive or partial recursive functions, theorem 2.1 below may be combined with theorem 2.4 of section 2.2.

The reason for making this restriction is twofold. For one thing, the consideration of arbitrary expressions made up from constants, variables and functions $f \in \mathcal{F}$ fits better into section 2.2, see definition 2.3. Secondly, there is no harm in requiring that $\mathcal{F}$ include a substantial number of functions and predicates, since the elements of $\mathcal{F}$ need not be listed explicitly. See the convention after the proof of theorem 2.3.

The *semantics* of $\mathcal{AL}(\mathcal{F})$-programs is for the most part obvious, but here again a few points require clarification.

Of course, execution starts with the first statement of the main program. The empty statement is executed by doing nothing. It is useful only with a label, at the end of the program: in this case it allows a jump to the end of the program, which terminates execution.

The execution of an assignment $Y := f(X_1, \ldots, X_m)$ is equivalent to the execution of a program computing f, hence it does not terminate if f is undefined for the current values of X_1, …, X_m. When a **for**-statement of the form given under (viii) is executed, then the values e_1, e_2 of E_1 and E_2 are computed at the beginning and stored; hence the number of executions is not effected by changes to E_1, E_2. When the execution is finished, the value of I is $e_2 + 1$ if $e_2 \geq e_1$, or e_1, if $e_2 < e_1$.

Finally, we consider calls of a procedure Pr. Pr is *recursive* iff it calls itself, either directly or indirectly. This means formally that there exist procedures P_0, …, P_m such that P_0 and P_m are both identical with Pr, and a call of P_i occurs in the body of the declaration of P_{i-1} for $i = 1, \ldots, m$.

Let Pr be declared as above. If Pr is nonrecursive, then the execution of a call $Pr(E_1, \ldots, E_k, U_1, \ldots, U_l)$ proceeds as follows: The formal value parameters $X_1, \ldots, X_k$ are initialized to the current values of E_1, …, E_k, then they are handled like local variables. All other local variables are initialized to 0. Afterwards S is executed with Y_1, …, Y_l replaced by U_1, …, U_l. If Pr is recursive, then this description is still essentially correct, but the execution of S may require executing another instance of Pr, which in turn may require the execution of a third etc. This complicates to some extend the definition of configurations, and it would also complicate the proof of the following theorems 2.1 and 2.2, if recursive procedures were considered. Actually, recursive procedures will be dealt with only much later (theorem 2.15).

The following exercise illustrates how to perform one step towards a complete formal definition of the semantics of $\mathcal{AL}(\mathcal{F})$, both operationally and denotationally.

Exercise 2.1:

(i) [2] Let R be some fixed set of registers. For an A-URM $U = (R, P)$ define $\Phi_U = \Phi_P$ analogously as it was done before and in theorem 1.1. If P consists of a single conditional **if** Q **then** S_1 **else** S_2, describe Φ_P in terms of Φ_{S_1} and Φ_{S_2}.

(ii) [2] If operational semantics is to be applied, then it is more natural to regard a conditional not as a single statement but as a sequence of statements. In this case a suitable syntax is

if Q **then**; S_1; **else**; S_2; **end**

where S_1 and S_2 are again sequences of statements. Extend definition 1.4 (ii) to the case where P contains such conditionals.

Remark 2.2: Strictly speaking $\mathcal{AL}(\mathcal{F})$ is not an extension of $\mathcal{WL}$, since the syntax of **while**-loops differs slightly. In order to transform a $\mathcal{WL}$-program P into an equivalent $\mathcal{AL}(\mathcal{F})$-progam $\tilde{P}$ one has to replace every **while**-loop **while** $X > 0$ **do**; S; **end** by **while** $X > 0$ **do begin** S **end**; moreover, $\mathcal{F}$ has to contain the functions $\lambda x.x+1$ and $\lambda x.x \dot{-} 1$ and the predicate $\lambda x.x > 0$. If P is special, then obviously the **while**-statements of $\tilde{P}$ are special.

Remark 2.3: We just used the attribute *equivalent* for programs P, P'. Its most rigorous meaning is that the execution of P has the same effect as the execution of P' on *all* variables. In other words: If $U = (R, P)$, $U' = (R, P')$, where R includes all variables of P and P', then in the notation of exercise 2.1 (i) we have $\Phi_U = \Phi_{U'}$. However, this is not how we will use this term later on. Instead it is understood that input- and output variables are specified in addition to the programs themselves, and the word *equivalent* means that P and P' compute the same function with respect to these variables. In fact, P and P' may have different sets of variables.

We are now ready for the main result of this section.

Theorem 2.1: Let f be computed by an $\mathcal{AL}(\mathcal{F})$-program P with respect to some tuple of variables, where P contains no recursive procedures.

(i) If every function $g \in \mathcal{F}$ is partial recursive and every predicate $Q \in \mathcal{F}$ is recursive, then f is partial recursive.

(ii) If in addition P contains no **goto**-statements and only *special* **while**-statements, and every function and predicate in $\mathcal{F}$ is primitive recursive, then f is primitive recursive.

We prove this theorem together with another result. It asserts that there is yet another control structure (besides **while** $X > 0$ **do**) which along with $\mathcal{WL}$-assignments suffices to simulate all $\mathcal{AL}$-programs:

Definition 2.1: (V) Call conditional jumps of the form

if $X > 0$ **then goto** L

restricted **goto**-*statements*. A $\mathcal{GL}$-*program* is an $\mathcal{AL}(\mathcal{F})$-program which contains no other instructions than $\mathcal{WL}$-assignments and restricted **goto**-statements. (A label may be attached in either case.)

Theorem 2.2: Every partial recursive function is computed by some $\mathcal{GL}$-program.

Proof of theorems 2.1 and 2.2: We show how to transform an $\mathcal{AL}$-program P into various programs P' which are equivalent to P, that is, each P' computes the same function f as P with respect to the same tuple of variables. More specifically, we show:

(I) If P satisfies the conditions from theorem 2.1 (ii), then P can be transformed into an equivalent special $\mathcal{WL}$-program P_1.

(II) If P satisfies the conditions from theorem 2.1 (i), then P can be transformed into an equivalent $\mathcal{GL}$-program P_2.

(III) P_2 (in fact, any $\mathcal{GL}$-program) can be transformed into an equivalent $\mathcal{WL}$-program P_3.

Now theorem 2.1 (ii) follows from (I), theorem 2.1 (i) from (II) and (III), and theorem 2.2 from (II) and remark 2.2. For the last claim note that the translation of a $\mathcal{WL}$-program into $\mathcal{AL}(\mathcal{F})$ trivially satisfies the conditions of theorem 2.1 (i).

Transformations (I) and (II) proceed in several steps; at every stage we denote the result of the steps preformed so far by $\hat{P}$. (Thus in the transformation algorithm $\hat{P}$ may be regarded as a program variable of type 'program' which is initialized to P.) $\hat{P}$ will always be an $\mathcal{AL}(\mathcal{F}')$-program for some $\mathcal{F}'$ which may contain a few additional functions and predicates besides those in $\mathcal{F}$.

Some steps of the transformations involve introducing a new variable V; such V must not occur in $\hat{P}$ before the step under consideration, and V must be different from all input- and output variables. In fact, the symbols V, V_1, V_2, ... automatically represent *new* variables. Of course, they do not serve as variable identifiers, but as metavariables representing such identifiers; see remark 1.1.

1. Let Pr be a procedure of $\hat{P}$ whose declaration does not in turn contain any local procedure declarations (if such Pr exists). The declaration of Pr has the form

> **procedure** $Pr(X_1, \ldots, X_k; \textbf{var } Y_1, \ldots, Y_l)$;
> **local** $Z_1, \ldots, Z_j$;
> **begin** S **end**

Now replace every procedure call $Pr(E_1, \ldots, E_k, U_1, \ldots, U_l)$ by the following sequence of statements, where S' is obtained from S by replacing every occurrence of Y_i by U_i:

> $X_1 := E_1; \ldots; X_k := E_k;$
> $Z_1 := 0; \ldots; Z_j := 0;$
> S'

In some obvious cases these statements have to be delimited by **begin** ... **end**, for instance, if the procedure call made up the body of a **while**-loop. More generally, it may be necessary to introduce **begin** ... **end** whenever a step of the program transformation involves the replacement of a single statement by a sequence of several statements (though this fact will not be mentioned later on). The translation just described can actually be used to provide a precise definition of the semantics of procedure calls.

The condition that all procedures are nonrecursive guarantees that this step really eliminates all calls of Pr; also, all remaining procedures are still nonrecursive afterwards. Repeat this step until all procedure calls of $\hat{P}$ are eliminated. Then cancel all procedure declarations from $\hat{P}$.

2. Replace every **for**-statement **for** $I := E_1$ **to** E_2 **do** S by the following sequence of statements:

$V_1 := E_1$; $V_2 := E_2$;
$V_2 := V_2 + 1$; $V_2 := V_2 \dot{-} V_1$; $I := V_1$;
while $V_2 > 0$ **do**
    begin $V_2 := V_2 \dot{-} 1$; S; $I := I + 1$ **end**

The new variables V_1 and V_2 are chosen *differently* for each **for**-statement (though in some cases the same V_1, V_2 could be used more than once). In subsequent steps it is also assumed that new variables V, V' etc. are chosen differently for each replacement, unless it is stated otherwise. The statement $V_2 := V_2 \dot{-} V_1$ is of the type $Y := g(X_1, \ldots, X_m)$ with g primitive recursive (exercise 1.4 (ii)); it will be eliminated in step 6.

If S was delimited by **begin** ... **end**, then these delimiters may now eliminated, since they are no longer needed; this remark also applies to some of the following steps. However, the transformation also works properly if redudant delimiters are left for the time being; they will be removed in steps 9 and 12.

3. Replace every **while**-statement **while** $Q(X_1, \ldots, X_m)$ **do** S with $Q(X_1, \ldots, X_m)$ not of the form $X > 0$ as follows:

$V := c_Q(X_1, \ldots, X_m)$;
while $V > 0$ **do**
    begin S; $V := c_Q(X_1, \ldots, X_m)$ **end**

Now the **while**-condition for each loop is of the form allowed in $\mathcal{WL}$-programs.

4. Replace every **case**-statement

case I **of** E_1: S_1; ... E_n: S_n; **else** S

by a nested conditional statement as follows:

$V_1 := E_1$; ...; $V_n := E_n$;
if $I = V_1$ **then** S_1
else ...
else if $I = V_n$ **then** S_n
else S

Recall that the predicate $\lambda xy.x = y$ is primitive, as shown in exercise 1.11 (iii).

5. Replace every conditional statement **if** $Q(X_1, \ldots, X_m)$ **then** S_1 **else** S_2 by the following sequence of statements:

$V_1 := c_Q(X_1, \ldots, X_m)$; $V_2 := 1$;
while $V_1 > 0$ **do**
    begin $V_1 := V_1 \dot{-} 1$; $V_2 := V_2 \dot{-} 1$; S_1 **end**;
while $V_2 > 0$ **do**
    begin $V_2 := V_2 \dot{-} 1$; S_2 **end**

The statement $V_2 := 1$ is again of the type $Y := g(X_1, \ldots, X_m)$, where $m = 0$ and $g = \lambda.1$; this function was shown to be primitive recursive in example 1.4 (i). Conditional statements of the form **if** $Q(X_1, \ldots, X_m)$ **then** S_1 are replaced similarly, but the second **while**-loop and the assignments to V_2 are no longer needed.

6. Next we eliminate statements of the form $Y := g(X_1, \ldots, X_m)$, except those where $g = \lambda x.x$ ($m = 1$), and $\mathcal{WL}$-assignments. Let S be a $\mathcal{WL}$-program which computes g with respect to new variables V_1, …, V_m, V, such that S is special if g is primitive recursive. Transform S into an $\mathcal{AL}(\mathcal{F}')$-program $\tilde{S}$ as described in remark 2.2. By renaming the variables of $\tilde{S}$ it may be assumed that neither input- and output-variables, nor any other variable of $\tilde{S}$ occurs in $\hat{P}$. Replace $Y := g(X_1, \ldots, X_m)$ as follows:

$V_1 := X_1; \ldots; V_m := X_m;$
$\tilde{S};$
$Y := V$

Note that this avoids undesired side effects. In particular, the arguments X_i of g are not changed, unless Y is one of them, even if S contains assignments to its input variables V_i.

7. Replace every occurrence of the empty statement or of assignments of the form $X := X$ by $V := V + 1$. (Here the same *dummy* variable V may be used for each replacement.)

8. Replace every assignment of the form $Y := X$, where X and Y are *different* variables, by the following sequence of statements:

while $Y > 0$ **do** $Y := Y \dot{-} 1$;
while $X > 0$ **do**
    begin $X := X \dot{-} 1$; $Y := Y + 1$; $V := V + 1$ **end**;
while $V > 0$ **do**
    begin $V := V \dot{-} 1$; $X := X + 1$ **end**

The first **while**-loop sets Y to 0, the second assigns the value of X to Y and to V, and the third restores the previous value of X. Recall that V is a newly introduced variable which, by our conventions, automatically has the value 0 at the beginning of the program. Hence there is no need to include the statement **while** $V > 0$ **do** $V := V \dot{-} 1$. Again the same variable V could be used for each replacement, since V again has the value 0 after the loop **while** $V > 0$ **do** ….

9. Now assume that the conditions of part (ii) of the theorem were satisfied by the original program P. Then the current program $\hat{P}$ contains only special **while**-statements as control structures. This is true since all control structures except for **while** were eliminated, and only special **while**-statements were newly introduced in the preceding steps. Also, all assignment statements are now of the form $X := X + 1$ or $X := X \dot{-} 1$. In order to convert $\hat{P}$ into a special $\mathcal{WL}$-program eliminate all statement-labels (which were in fact useless from the beginning) and all **begin** … **end**-pairs except for those which delimite the body of some **while**-loop. Then replace every **while**-statement **while** $X > 0$ **do begin** S **end** by **while** $X > 0$ **do**; S; **end**.

At this point the desired program P_1 has been constructed. This completes the proof of claim (I), and hence of assertion (ii) of the theorem.

The remaining steps 10–14 serve to prove assertions (II) and (III), and they are carried out *instead of* step 9. Of course it is no longer assumed that P necessarily satisfies the conditions from part (ii) of the theorem, but only those of part (i). In this case $\mathcal{WL}$-assignments, compound statements, **while**-statements of the form **while** $X > 0$ **do** ... and unconditional jumps may be left after step 8.

10. Insert the assignment $V := V + 1$ at the beginning of $\hat{P}$, where V is a new variable as usual. No other assignments to V will be introduced, hence V will have value 1 throughout. Then replace every unconditional jump **goto** L by the restricted **goto**-statement **if** $V > 0$ **then goto** L.

11. Let V be the same variable which was introduced in step 10. Replace every **while**-statement **while** $X > 0$ **do** S as follows:

$$\begin{array}{ll} & \textbf{if } V > 0 \textbf{ then goto } L_2; \\ L_1: & S; \\ L_2: & \textbf{if } X > 0 \textbf{ then goto } L_1; \end{array}$$

If the statement S already had a label L before this step was performed, then it may be preserved, that is, L_1 may be chosen equal to L. Otherwise L_1 has to be a new label which did not occur in $\hat{P}$ before. L_2 must a new label in any case.

12. Since restricted **goto**-statements are the only control structure left, the delimiters **begin** ... **end** are now redudant and may be eliminated. This may leave extra labels for some statements, which have to be eliminated as well. For instance, if the compound statement

$$L_1 : \textbf{begin } L_2 : S \textbf{ end}$$

occurs, then replace it by $L_2 :\ S$, and replace **goto** L_1 by **goto** L_2 throughout.

The result of this step is the desired $\mathcal{GL}$-program P_2, completing the proof of claim (II).

13. Now comes the tricky part: we transform P_2 into an equivalent $\mathcal{AL}$-program P_3' without **goto**-statements. From P_3' the desired $\mathcal{WL}$-program P_3 will be obtained in a straightforward way. In steps 1–12 we proceeded by local translations, that is, each individual instruction of a certain undesired type was replaced by a different kind of construct. In the present step this method does not succeed, however.

The following idea is applied instead. Let P_2 be of the form $I_0; \ldots; I_{l-1}$. The values of the variables of P_3' represent a configuration $\mathcal{C}$ of P_2. Hence P_3' has a new variable, this time denoted by C, which represents the instruction counter. P_3' has the form

$$\textbf{while } Q \textbf{ do } S,$$

where each execution of S corresponds to one step of a computation Γ by P_2, and Q holds if and when a halting configuration is reached. Thus Q is the condition $C < l$, which is primitive recursive by exercise 1.11 (i). The work performed in one step of Γ depends on the instruction I_i, where i is the current value of the instruction counter. Accordingly S, the body of the main loop of P_3', has the form

$$\textbf{case } C \textbf{ of } 0{:}\, S_0;\ \ldots\ l-1{:}\ S_{l-1};\ \textbf{else},$$

where S_i simulates instruction I_i of P_2. If I_i is of the form $X := X + 1$, then S_i is the statement

$$\textbf{begin } X := X + 1; C := C + 1 \textbf{ end},$$

and similarly if I_i is of the form $X := X \dot{-} 1$. On the other hand, if I_i is of the form **goto** L, then S_i is the statement $C := k$, where I_k is the instruction of P_2 with label L. (Recall that $\lambda x.k$ is primitive recursive by exercise 1.4 (i).) Note that for any input the initial values of the variables of P_3' represent the corresponding initial configuration of P_2, since C is automatically initialized to 0.

14. Finally, repeat steps 3–6 and 8, which leaves **while**-statements of the form **while** $X > 0$ **do** as the only control structure. The $\mathcal{WL}$-program P_3 is obtained according to remark 2.2.

This completes the the proof of (III) and of theorems 2.1 and 2.2. □

Exercise 2.2: Show that the following functions and predicates are primitive recursive:

(i) [2] the predicates $\leq$, $>$, $\geq$ and $\neq$;

(ii) [2] the functions $\lambda xy.\lfloor x/y \rfloor$ and $\lambda xy.x \bmod y$; $\lfloor x/0 \rfloor$ and $x \bmod 0$ are defined to be 0 and x respectively.

From a practical point of view, theorem 2.1 (ii) is unsatisfactory inasmuch as programs in real life often contain **while**-statements which are not special, even though the function computed is almost always primitive recursive. However, the requirement that **while**-statements be special can be replaced by a less stringent condition; this is the contents of our next result. Roughly speaking, it suffices to guarantee that **while**-loops are not executed 'too often.'

Let I be the **while**-statement **while** Q **do** S in an $\mathcal{AL}$-program P, U an A-URM (R, P), and let V be a mapping from R to ω. By $\#_I(V)$ we denote the number of executions of the body S before the **while**-loop is left, assuming that every variable X has the value $V(X)$ prior to execution. This could be formalized using operational semantics for $\mathcal{AL}$ (which we did not define). Instead we provide the following example as an illustration.

Example 2.1: Let I be the statement

> **while** $X > 0$ **do**
> **begin if** $Y = 0$ **then goto** L; $Y := Y \dot{-} 1$; $X := X \dot{-} 1$; $X := X \dot{-} 1$ **end**,

S the body of I; the label L occurs outside I. Also, let $V(X) = x$, $V(Y) = y$. If y is sufficiently large (in comparison with x), then S is executed $\lceil x/2 \rceil$ times, after which X has the value 0. If x is sufficiently large, then there will be $y + 1$ executions of S. (The last consists only in the jump to L, but it counts all the same.) Thus

$$\#_I(V) = \min(\lceil x/2 \rceil, y + 1).$$

Definition 2.2: (N) With notation as above let $Y_1, \ldots, Y_k$ be variables of U, f a function. The **while**-statement I is *bounded by* $f(Y_1, \ldots, Y_k)$, iff for all V $\#_I(V) \leq f\big(V(Y_1), \ldots, V(Y_k)\big)$.

Theorem 2.3: Let P be an $\mathcal{AL}(\mathcal{F})$-program without jumps and recursive procedures, where every function and predicate in $\mathcal{F}$ is primitive recursive, and every **while**-statement is bounded by an expression of the form $f(Y_1, \ldots, Y_k)$, with f primitive recursive. Then the function computed by P with respect to any given tuple of variables is primitive recursive.

Proof: As before let I be a **while**-statement of P of the form **while** Q **do** S, bounded by $f(Y_1, \dots, Y_k)$ with $f \in \mathcal{PR}$. Replace I by the following sequence of statements I', where Z and K are new variables:

$Z := f(Y_1, \dots, Y_k)$;
for $K := 1$ **to** Z **do**
  if Q **then** S

Assume the execution of I' starts while every variable X has value $V(X)$. Then the first $\#_I(V)$ executions of the body of the **for**-loop simulate the execution of I, and the remaining ones have no effect. Thus I' is indeed equivalent to I. By making this replacement for every **while**-statement of P an equivalent program P' is obtained which satisfies the conditions of theorem 2.1 (ii). The assertion follows. □

From now on we will often write $\mathcal{AL}$ for $\mathcal{AL}(\mathcal{F})$; it is understood that this means $\mathcal{AL}(\mathcal{F})$ for some set of (probably simple) functions and predicates $\mathcal{F}$. For instance, when we say that a certain $\mathcal{AL}$-program P satisfies the criteria of theorem 2.1 (ii) above, then this implies that P actually belongs to $\mathcal{AL}(\mathcal{F})$, where $\mathcal{F}$ contains only primitive recursive functions and predicates.

Example 2.2: The following program P, which represents the well-known Euclidean algorithm, computes $\lambda xy.\, \mathrm{gcd}(x, y)$ with respect to X, Y and X:

while $Y > 0$ **do**
  begin $Z := X \bmod Y$; $X := Y$; $Y := Z$ **end**

Here every execution of the body of the **while**-loop leaves $\mathrm{gcd}(X, Y)$ unchanged, since $\mathrm{gcd}(x, y) = \mathrm{gcd}(y, x \bmod y)$. If the original values of X and Y are x and y, and x' is the value of X upon termination, then $\mathrm{gcd}(x, y) = \mathrm{gcd}(x', 0) = x'$. The proof that the program terminates is based on the fact that every execution of the body of the loop reduces the value of Y, since $x \bmod y < y$ for $y > 0$. Obviously the **while**-statement is bounded by Y, hence the function gcd is primitive recursive.

Exercise 2.3: [4] In an $\mathcal{AL}$-program let I be a conditional or unconditional jump with destination L, I' the statement with label L. Call I an *outward jump*, iff I' does not occur in the body of the same procedure as I, a *forward jump*, if I is not an outward jump and I' occurs after I, a *backward jump*, if I is not an outward jump and I' is identical with or occurs before I. Show that the conclusion in theorem 2.1 (ii) and in theorem 2.3 remains valid if P contains forward jumps (but no outward or backward jumps).

2.2. Definition schemes for primitive recursive and partial recursive functions.

The main purpose of this section is to show that the classes $\mathcal{PR}$, $\mathcal{R}$ and $\mathcal{P}$ are closed under various operations. Most theorems are of the following form: If g is obtained from (primitive/partial) recursive functions f_1, f_2, ... in a certain way, then g is again (primitive/partial) recursive. This means: If f_1, f_2, ... are primitive recursive, then so is g, and the same assertion holds with 'primitive recursive' replaced by 'recursive' or 'partial recursive.' In addition, many more specific functions and predicates are shown to be primitive recursive.

Primitive recursiveness has already been established for the functions $\lambda xy.x+y$, $\lambda xy.x \dot{-} y$, $\lambda xy.xy$, $\lambda xy.\lfloor x/y \rfloor$ and $\lambda xy.x \bmod y$, as well as the predicates $<$, $\leq$ and $=$ among others (examples 1.1 and 1.2, exercises 1.4, 1.11 and 2.2). The first result of this section shows that the same can be deduced for functions and predicates made up from them, like $\lambda xyz.\lfloor (x+y)/z \rfloor$ or $\lambda xyz.(x+y)(x \dot{-} z) < xz + y + z$.

Definition 2.3: (V) Let $\mathcal{V}$ be a set of variables, $\mathcal{F}$ a set of functions in $\mathcal{NF}$ and predicates in $\mathcal{NS}$.

(i) The set of all *numerical* $(\mathcal{V}, \mathcal{F})$*-expressions* is the smallest set $\mathcal{M}$ such that
   (a) Every element of $\mathcal{V}$ and every constant $c \in \omega$ belongs to $\mathcal{M}$.
   (b) If $t_1, \ldots, t_m$ are elements of $\mathcal{M}$ and f is an m-place function in $\mathcal{F}$, then $f(t_1, \ldots, t_m) \in \mathcal{M}$.

(ii) The set of all *Boolean* $(\mathcal{V}, \mathcal{F})$*-expressions* is the smallest set $\mathcal{B}$ such that
   (a) The constants **true** and **false** belong to $\mathcal{B}$.
   (b) If $t_1, \ldots, t_m$ are numerical $(\mathcal{V}, \mathcal{F})$-expressions and P is an m-place predicate in $\mathcal{F}$, then $P(t_1, \ldots, t_m) \in \mathcal{B}$.
   (c) If b, $b' \in \mathcal{B}$, then $\neg b$, $(b \,\&\, b')$, $(b \vee b')$, $(b \Rightarrow b')$ and $(b \Leftrightarrow b')$ are elements of $\mathcal{B}$.

For the moment denote the set of numerical $(\mathcal{V}, \mathcal{F})$-expressions by $\mathcal{M}_0$. The phrase '$\mathcal{M}_0$ is the smallest set $\mathcal{M}$ satisfying (a) and (b)' can be made precise by observing that every element of $\mathcal{M}_0$ may be regarded as a finite sequence of elements of Σ, where Σ contains the elements of $\mathcal{V}$ and $\mathcal{F}$, as well as digits, left and right parenthesis and commas. If we denote the set of all such sequences by Σ^* (see the paragraph preceding definition 2.11), then $\mathcal{M}_0$ can be defined as the *intersection* of all subsets of Σ^* satisfying conditions (a) and (b) of definition 2.3 (i). For part (ii) of this definition one can proceed similarly, of course. We note that the notation Σ^* is most commonly used for *finite* Σ, as in definition 2.12 (v).

When expressions are actually written down, then we may of course deviate from definition 2.3, using infix notation for binary operations and other mathematical conventions. For example, we may write $(x+y)^2 =$ as an abbreviation for the $(\{x, y\}, \{f_\times, f_+\})$-expression $f_\times\big(f_+(x, y), f_+(x, y)\big)$, where f_+ and $f_\times$ are the functions $\lambda xy.x+y$ and $\lambda xy.xy$. Also, unnecessary parenthesis may be omitted in Boolean expressions, using the convention that $\neg$ has higher precedence than & and $\vee$, and & and $\vee$ have higher precedence than $\Rightarrow$ and $\Leftrightarrow$.

In order to prove an assertion $A(t)$ for every numerical $(\mathcal{V}, \mathcal{F})$-expression t, it suffices to show that $A(t)$ holds for every variable and every constant t, and that $A(t_1)$, $\ldots$, $A(t_m)$ implies $A\big(f(t_1, \ldots, t_m)\big)$ for $f \in \mathcal{F}$. Similarly, an assertion $B(b)$ is shown for every Boolean $(\mathcal{V}, \mathcal{F})$-expression b by proving that it holds for every b of the form **true**, **false** or $P(t_1, \ldots, t_m)$, and that $B(b)$ and $B(b')$ imply $B(\neg b)$, $B(b \,\&\, b')$ etc. This amounts to induction on the length of t or b, where t or b is regarded as an element of Σ^* as above.

If $\mathcal{V} = \{x_1, \ldots, x_n\}$, and every function in $\mathcal{F}$ is total, then every numerical $(\mathcal{V}, \mathcal{F})$-expression t and every Boolean $(\mathcal{V}, \mathcal{F})$-expression b represent a total function $\lambda \vec{x}_n.t$ and a predicate $\lambda \vec{x}_n.b$ in the obvious way. If we drop the requirement that the functions in $\mathcal{F}$ are total, then $\lambda \vec{x}_n.t$ is still a well-defined partial function; the convention is that the value of t is undefined ($t\uparrow$) whenever that of some subexpression of t is undefined. For instance, if $g(x)\uparrow$, then $0 \cdot g(x)\uparrow$, even though $0 \cdot x = 0$ for any number x.

On the other hand, a predicate P will not be applied to expressions $t_1, \ldots, t_m$, unless all t_i are defined. An exception is the predicate $=$; $t_1 = t_2$ means that either t_1 and t_2 are both defined and have the same value, or both are undefined. Thus a function f may be defined by an equation of the form $f(\vec{x}_n) = t$; this includes the case where t is undefined for some $x_1, \ldots, x_n$. The following definition and theorem, however, only deal with Boolean expressions which are made up from total functions.

Definition 2.4: A function f is defined from a set of functions $\mathcal{F}$ by *explicit definition*, iff f is of the form $\lambda\vec{x}_n.t$ for some numerical $(\mathcal{V}, \mathcal{F})$-expression t, where $\mathcal{V} = \{x_1, \ldots, x_n\}$. Similarly, a predicate P is defined from a set of *total* functions and predicates $\mathcal{F}$ by explicit definition, iff P is of the form $\lambda\vec{x}_n.b$ for some Boolean $(\mathcal{V}, \mathcal{F})$-expression b, again with $\mathcal{V} = \{x_1, \ldots, x_n\}$.

Theorem 2.4: Let $\mathcal{F}$ be a set of functions and predicates, f a function and P a predicate defined from $\mathcal{F}$ by explicit definition.

(i) If every function in $\mathcal{F}$ is (primitive/partial) recursive, then f is (primitive/partial) recursive.

(ii) If every function and predicate in $\mathcal{F}$ is (primitive) recursive, then P is (primitive) recursive.

Proof: (i) If t is a variable or constant, then the assertion follows from exercises 1.2 (i) and 1.4 (i). Now let t be of the form $f(t_1, \ldots, t_m)$, assume that the assertion holds for all t_i, and let $g = \lambda\vec{x}_n.t$, $g_i = \lambda\vec{x}_n.t_i$. Then the following program P computes g with respect to $X_1, \ldots, X_n, Y$:

$$
\begin{array}{l}
Z_1 := g_1(X_1, \ldots, X_n); \\
\quad\vdots \\
Z_m := g_m(X_1, \ldots, X_n); \\
Y := f(Z_1, \ldots, Z_m)
\end{array}
$$

If all functions $f \in \mathcal{F}$ are partial recursive, then all g_i are partial recursive by induction hypothesis, and it follows by theorem 2.1 (i) that g is partial recursive. If all $f \in \mathcal{F}$ are recursive, then all g_i are recursive, hence g is partial recursive and total, that is, recursive. Finally, if all $f \in \mathcal{F}$ are primitive recursive, then the program P safisfies the conditions of theorem 2.1 (ii) (since f and all g_i are primitive recursive), hence g is primitive recursive.

(ii) Let Q be the predicate $\lambda\vec{x}_n.b$. The case where b is of the form **true** or **false** has been dealt with in example 1.4 (i). If b is of the form $P(\vec{t}_m)$, where the t_i are numerical $(\mathcal{V}, \mathcal{F})$-expressions, then $c_Q(\vec{x}_n) = c_P(\vec{t}_m)$, hence the assertion follows from part (i). Now assume that it holds for b_1 and b_2. For the rest of the proof it suffices to show that the assertion also holds for $\neg b_1$ and $b_1 \mathbin{\&} b_2$, since every logical connective can be expressed in terms of $\neg$ and &. Let Q_i be the predicate $\lambda\vec{x}_n.b_i$ $(i = 1, 2)$. Then

$$
c_Q(\vec{x}_n) = \begin{cases} 1 \dot{-} c_{Q_1}(\vec{x}_n) & \text{if } b = \neg b_1, \\ c_{Q_1}(\vec{x}_n)c_{Q_2}(\vec{x}_n) & \text{if } b = b_1 \mathbin{\&} b_2. \end{cases}
$$

Once again the assertion follows from (i). The proof is complete. $\square$

Example 2.3: (i) For every n the function $\lambda\vec{x}_n.x_1+\cdots+x_n$ is primitive recursive, since it is defined from $\lambda xy.x+y$ by explicit definition. The same is true for other operations which can be applied to any number of arguments, and which are primitive recursive for two arguments, like multiplication, gcd (example 2.2) or max and min (which will be considered in exercise 2.5 below).

(ii) Every polynomial function $p(\vec{x}_n)$ with nonnegative integer coefficients is primitive recursive, since it is defined by explicit definition from addition and multiplication.

(iii) Theorem 2.4 (ii) implies, among other things, that the recursive and the primitive recursive sets and predicates are closed under all Boolean operations. This yields another solution for exercise 2.2 (i), since $\leq$ is the disjunction of $<$ and $=$, dealt with in exercise 1.11, and $\geq$, $>$ and $\neq$ are the negations of $<$, $\leq$ and $=$. Also, if $f_1(\vec{x}_n), \ldots, f_m(\vec{x}_n)$ are (primitive) recursive, then so is the predicate $\lambda\vec{x}_n.f_1(\vec{x}_n) < \cdots < f_m(\vec{x}_n)$: Just write it in the form $\lambda\vec{x}_n.\big(f_1(\vec{x}_n) < f_2(\vec{x}_n) \,\&\cdots\&\, f_{m-1}(\vec{x}_n) < f_m(\vec{x}_n)\big)$. The same conclusion holds if some occurrences of the symbols $<$ are replaced by $\leq$ or $=$, or if all are replaced by $>$, $\geq$ or $=$.

(iv) If $f(\vec{x}_n)$ is (primitive) recursive, then so is the set $\mathrm{gr}(f)$, which corresponds to the predicate $\lambda\vec{x}_n y.f(\vec{x}_n) = y$. An interesting question is, to what extend the converse of this fact holds. It is easy to see that it does not hold without restriction, since $\mathrm{gr}(f)$ may be primitive recursive for non-total f, for instance, if $f(\vec{x}_n)$ is nowhere defined. This question will be answered in theorem 3.13.

Exercise 2.4: (i) [2] Generalize example 2.3 (ii) by showing that a polynomial function $p(\vec{x}_n)$ is still primitive recursive if the coefficients are arbitrary rational numbers, but $p(\vec{x}_n) \in \omega$ for all $\vec{x}_n$. An example is $p(x) = \frac{1}{2}x^2 - \frac{1}{2}x$.

(ii) [1] Show that $\lambda xy.|x-y|$ is primitive recursive.

The next theorem concerns functions defined by *cases*.

Theorem 2.5: Let $Q_1(\vec{x}_n), \ldots, Q_k(\vec{x}_n)$ be predicates such that for every $\vec{x}_n$ exactly one of the assertions $Q_i(\vec{x}_n)$ $(1 \leq i \leq k)$ holds, let $f_1(\vec{x}_n), \ldots, f_k(\vec{x}_n)$ be functions, and let f be defined by

$$f(\vec{x}_n) = \begin{cases} f_1(\vec{x}_n) & \text{if } Q_1(\vec{x}_n), \\ \quad\vdots & \\ f_k(\vec{x}_n) & \text{if } Q_k(\vec{x}_n). \end{cases}$$

If the functions f_i and the predicates Q_i are (primitive) recursive, then f is (primitive) recursive. If the Q_i are recursive and the f_i are partial recursive, then f is partial recursive.

Proof: The function f is computed with respect to $X_1, \ldots, X_n, Y$ by the following program:

if $Q_1(X_1,\ldots,X_n)$ **then** $Y := f_1(X_1,\ldots,X_n)$
else if $Q_2(X_1,\ldots,X_n)$ **then** $Y := f_2(X_1,\ldots,X_n)$
$\vdots$
else if $Q_{k-1}(X_1,\ldots,X_n)$ **then** $Y := f_{k-1}(X_1,\ldots,X_n)$
else $Y := f_k(X_1,\ldots,X_n)$

By theorem 2.1 f is primitive recursive if all Q_i and f_i are primitive recursive, or partial recursive, if the Q_i are recursive and the f_i are partial recursive. If all Q_i and f_i are recursive, then f is partial recursive and total, that is, f is recursive. □

Exercise 2.5: [0] Show that $\lambda xy.\max(x, y)$ and $\lambda xy.\min(x, y)$ are primitive recursive.

Exercise 2.6: [2] (i) Let f, $g\colon \omega^n \to \omega$ be such that $g(\vec{x}_n) = f(\vec{x}_n)$ for almost all $\vec{x}_n$. Show that f is partial recursive if and only if g is partial recursive. Also, if f and g are total, then f is (primitive) recursive if and only if g is (primitive) recursive.

(ii) Show that if $f\colon \omega \to \omega$ is eventually periodic, that is, there exist k and l such that $f(x + l) = f(x)$ for all $x \geq k$, then f is primitive recursive.

Definition 2.5: (N) (i) A set of functions $\mathcal{F} \subseteq \mathcal{NF}$ is *invariant under finite changes*, iff $f \in \mathcal{F} \iff g \in \mathcal{F}$ holds for any two functions f, g such that $f(\vec{x}_n) = g(\vec{x}_n)$ for almost all $\vec{x}_n$.

(ii) If $\mathcal{F} \subseteq \mathcal{NF}$ contains only total functions, then $\mathcal{F}$ is *t-invariant under finite changes*, iff $f \in \mathcal{F} \iff g \in \mathcal{F}$ holds for any two *total* functions f, g such that $f(\vec{x}_n) = g(\vec{x}_n)$ for almost all $\vec{x}_n$.

(iii) A set of sets $\mathcal{S} \subseteq \mathcal{NS}$ is *invariant under finite changes*, iff $A \in \mathcal{S} \iff B \in \mathcal{S}$ holds for any two sets A, B whose symmetrical difference $A \mathbin{\triangle} B = (A - B) \cup (B - A)$ is finite.

Thus exercise 2.6 shows that $\mathcal{P}$ is invariant under finite changes, and that $\mathcal{PR}$ and $\mathcal{R}$ are t-invariant under finite changes. Applying this fact to characteristic functions, it follows that $\mathcal{PR}_*$ and $\mathcal{R}_*$ are also invariant under finite changes. Finally, if $A \mathbin{\triangle} B$ is finite, and A is the domain of a partial recursive function f, then a partial recursive function with domain B is obtained by changing f at finitely many places. Hence if A is r.e., then B is r.e., and by symmetry the converse implication also holds. Hence $\mathcal{E}$ is also invariant under finite changes.

Certainly the most important method for constructing new functions are *recursive definitions*. This means that f is defined by expressing $f(x_1, \ldots, x_n)$ in terms of one or several other values $f(x'_1, \ldots, x'_n)$, $f(x''_1, \ldots, x''_n)$ .... Also, two functions f, g may be defined by *simultaneous* recursion, which means that $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$ are expressed in terms of some of the values of both f and g.

At the moment we are not yet in a position to handle recursion in general, but only a rather special case.

Definition 2.6: (i) A function $f(x, \vec{y}_n)$ is defined from $g(\vec{y}_n)$ and $h(x, \vec{y}_n, z)$ by *primitive recursion*, iff

$$\begin{aligned} f(0, \vec{y}_n) &= g(\vec{y}_n), \\ f(x + 1, \vec{y}_n) &= h\big(x, \vec{y}_n, f(x, \vec{y}_n)\big). \end{aligned}$$

(ii) More generally, functions $f_1(x, \vec{y}_n)$, ..., $f_k(x, \vec{y}_n)$ are defined from $g_1(\vec{y}_n)$, ..., $g_k(\vec{y}_n)$ and $h_1(x, \vec{y}_n, \vec{z}_k)$, ..., $h_k(x, \vec{y}_n, \vec{z}_k)$ by *simultaneous primitive recursion*, iff

$$\left.\begin{aligned} f_i(0, \vec{y}_n) &= g_i(\vec{y}_n), \\ f_i(x + 1, \vec{y}_n) &= h_i\big(x, \vec{y}_n, f_1(x, \vec{y}_n), \ldots, f_k(x, \vec{y}_n)\big). \end{aligned}\right\} \quad i = 1, \ldots, k$$

(The term *primitive recursion* without further specification only applies to the definition scheme from (i); (ii) reduces to it for $k = 1$.)

Theorem 2.6: If $f_1, \ldots, f_k$ are defined from (primitive/partial) recursive functions $g_1, \ldots, g_k, h_1, \ldots, h_k$ by simultaneous primitive recursion, then the functions f_i are (primitive/partial) recursive.

Proof: Assume first that the functions g_i, h_i are total, so that the functions f_i will also be total. The following program computes f_i with respect to $X, Y_1, \ldots, Y_n, Z_i$ for each i; Z_i is assigned the values $f_i(0, \vec{y}_n), \ldots, f_i(x, \vec{y}_n)$ in order.

$$
\begin{array}{l}
Z_1 := g_1(Y_1, \ldots, Y_n) \\
\quad\vdots \\
Z_k := g_k(Y_1, \ldots, Y_n) \\
\textbf{for } I := 1 \textbf{ to } X \textbf{ do} \\
\qquad \textbf{begin} \\
\qquad\qquad U_1 := Z_1; \\
\qquad\qquad\quad \vdots \\
\qquad\qquad U_k := Z_k; \\
\qquad\qquad Z_1 := h_1(I \dot{-} 1, Y_1, \ldots, Y_n, U_1, \ldots, U_k); \\
\qquad\qquad\quad \vdots \\
\qquad\qquad Z_k := h_k(I \dot{-} 1, Y_1, \ldots, Y_n, U_1, \ldots, U_k) \\
\qquad \textbf{end}
\end{array}
$$

Thus in this case the assertion follows as in theorems 2.4 and 2.5. The case where the functions g_i, h_i are not necessarily total is dealt with in exercise 2.7 (i). □

Theorem 2.6 is most commonly used with $k = 1$, that is, primitive recursion is used much more often than simultaneous primitive recursion of several functions. This recursion scheme is closely connected with the concept of primitive recursive function, as will be seen in theorem 4.1.

Remark 2.4: Assume that new functions f (f_i) are constructed analogously as in definition 2.6, but by recursion on some other argument than the first. Then the conclusion from theorem 2.6 trivially remains valid, although, strictly speaking, we have deviated from the scheme of (simultaneous) primitive recursion.

Remark 2.5: A second point is more subtle. Definition 2.6 stipulates that all the variables x, y_i and z_i really occur as arguments in each h_j; call this requirement (α). As an illustration, consider the following definition of f_1, f_2:

$$
\begin{aligned}
f_1(0, y_1, y_2) &= g_1(y_1, y_2), \\
f_2(0, y_1, y_2) &= g_2(y_1, y_2), \\
f_1(x+1, y_1, y_2) &= h_1'\big(x, y_1, f_2(x, y_1, y_2)\big), \\
f_2(x+1, y_1, y_2) &= h_2'\big(x, y_2, f_1(x, y_1, y_2)\big).
\end{aligned} \tag{1}
$$

Here requirement (α) is violated, since y_2 and $f_1(x, y_1, y_2)$ do not occur as arguments of h_1' in the third line, and similarly for the forth line. This observation seems irrelevant at first glance; after all, it would appear that (1) is equivalent to the following simultaneous primitive recursion, where $h_1 = \lambda xy_1y_2z_1z_2.h_1'(x, y_1, z_2)$, $h_2 = \lambda xy_1y_2z_1z_2.h_2'(x, y_2, z_1)$:

$$\begin{aligned}
f_1(0, y_1, y_2) &= g_1(y_1, y_2),\\
f_2(0, y_1, y_2) &= g_2(y_1, y_2),\\
f_1(x+1, y_1, y_2) &= h_1\big(x, y_1, y_2, f_1(x, y_1, y_2), f_2(x, y_1, y_2)\big),\\
f_2(x+1, y_1, y_2) &= h_2\big(x, y_1, y_2, f_1(x, y_1, y_2), f_2(x, y_1, y_2)\big).
\end{aligned} \tag{2}$$

(1) and (2) are indeed equivalent if the functions g_i, h_i, and hence f_i, are total. But if this is not the case, then the right-hand side of line 3 or 4 of (1) may be defined for some x, y_1, y_2 where it is undefined in (2). For instance, if $f_1(x, y_1, y_2)\uparrow$, then $h_1'\big(x, y_1, f_2(x, y_1, y_2)\big)$ may be defined, although $h_1\big(x, y_1, y_2, f_1(x, y_1, y_2), f_2(x, y_1, y_2)\big)\uparrow$. As a consequence, the functions f_i defined by (1) may have larger domain than those defined by (2).

In specific examples requirement (α) will play no rôle, since (simultaneous) primitive recursion will be applied only to total functions. From a theoretical point of view it must be noted, though, that the proof of theorem 2.6 becomes more complicated if requirement (α) were dropped; see exercises 2.7 (ii) and 2.28.

Exercise 2.7: (i) [2] What goes wrong with the above proof of theorem 2.6, if the functions g_i, h_i are not necessarily total? Explain how it has to be modified.

(ii) [6] Show that theorem 2.6 remains valid if requirement (α) from remark 2.5 is dropped.

Example 2.4: (i) Let $f = \lambda xy.xy$. We have

$$\begin{aligned}
f(0, y) &= 0,\\
f(x+1, y) &= f(x, y) + y.
\end{aligned}$$

Thus f is defined by primitive recursion from $\lambda y.0$ and $\lambda xyz.z+y$. These two functions are primitive recursive by theorem 2.4 (i), hence the same follows for f. (This was already established in exercise 1.4 (iii).)

(ii) Consider the exponential function $g = \lambda xy.x^y$. We have

$$\begin{aligned}
g(x, 0) &= 1\\
g(x, y+1) &= x \cdot x^y.
\end{aligned}$$

Thus g is defined by recursion on the second rather than the first argument, but by remark 2.4 it can still be inferred that g is primitive recursive. Alternatively, we may first define $g' = \lambda xy.y^x$ by primitive recursion (without deviating from definition 2.6) and then obtain g by explicit definition: $g(x, y) = g'(y, x)$.

(iii) Let $f(x) = \lfloor\sqrt{x}\rfloor$. Then

$$\begin{aligned}
f(0) &= 0,\\
f(x+1) &= \begin{cases} f(x)+1 & \text{if } \big(f(x)+1\big)^2 = x+1,\\ f(x) & \text{if } \big(f(x)+1\big)^2 \neq x+1. \end{cases}
\end{aligned}$$

The right side of the equation for $f(x+1)$ may be written in the form $h\big(x, f(x)\big)$. Then h is primitive recursive by theorems 2.4 and 2.5, hence f is primitive recursive.

(iv) Let $g(x, \vec{y}_n)$ be a function, and let

$$f(x, \vec{y}_n) = \sum_{z=0}^{x-1} g(z, \vec{y}_n).$$

Then

$$\begin{aligned} f(0, \vec{y}_n) &= 0, \\ f(x+1, \vec{y}_n) &= g(x, \vec{y}_n) + f(x, \vec{y}_n). \end{aligned}$$

Hence if g is (primitive/partial) recursive, then so is f. The same is holds for functions like $\lambda x\vec{y}_n.\prod_{z=0}^{x-1} g(z, \vec{y}_n)$, $\lambda x\vec{y}_n.\max_{0\le z<x} g(z, \vec{y}_n)$ etc. Moreover, if $h(\vec{x}_m, \vec{y}_n)$ is also (primitive/partial) recursive, then the same is true for $\lambda\vec{x}_m\vec{y}_n.\sum_{z=0}^{h(\vec{x}_m,\vec{y}_n)-1} g(z, \vec{y}_n)$, since

$$\sum_{z=0}^{h(\vec{x}_m,\vec{y}_n)-1} g(z, \vec{y}_n) = f\big(h(\vec{x}_m, \vec{y}_n), \vec{y}_n\big).$$

Putting $h(x, \vec{y}_n) = x+1$, we obtain $\sum_{z=0}^{x} g(z, \vec{y}_n)$. Again, all this remains valid if sums are replaced by products or maxima etc.

(v) Let $Q(x, \vec{y}_n)$ be a predicate, and let R, R' be defined by

$$\begin{aligned} R(x, \vec{y}_n) &\equiv (\forall z < x)\big(Q(z, \vec{y}_n)\big), \\ R'(x, \vec{y}_n) &\equiv (\exists z < x)\big(Q(z, \vec{y}_n)\big). \end{aligned}$$

We say that R and R' are obtained from Q by *bounded quantification.* Now

$$c_R(x, \vec{y}_n) = \prod_{z=0}^{x-1} c_Q(z, \vec{y}_n),$$

hence if Q is (primitive) recursive, then so is R. The same is true for R', since

$$R'(x, \vec{y}_n) \equiv \neg(\forall z < x)\big(\neg Q(z, \vec{y}_n)\big),$$

and we know that the class of (primitive) recursive predicates is closed under negation and bounded universal quantification. As in (iv) we can also conclude: If the predicate $Q(x, \vec{y}_n)$ and the function $h(\vec{x}_m, \vec{y}_n)$ are (primitive) recursive, then so are $\lambda\vec{x}_m\vec{y}_n.\big(\forall z < h(\vec{x}_m, \vec{y}_n)\big)\big(Q(z, \vec{y}_n)\big)$ and $\lambda\vec{x}_m\vec{y}_n.\big(\exists z < h(\vec{x}_m, \vec{y}_n)\big)\big(Q(z, \vec{y}_n)\big)$. Putting $h(x, \vec{y}_n) = x + 1$ we obtain bounded quantification of the form $(\forall z \le x)(\ldots)$ or $(\exists z \le x)(\ldots)$.

(vi) If $f\colon \omega \to \omega$ is (primitive/partial) recursive, then so is $g = \lambda xy.f^x(y)$, since

$$\begin{aligned} g(0, y) &= y, \\ g(x+1, y) &= f\big(g(x, y)\big). \end{aligned}$$

We say that g is obtained from f by (*pure*) *iteration.*

(vii) Here is an example for *simultaneous* primitive recursion. The *Fibonacci numbers* F_i are defined by

$$F_0 = 0, \quad F_1 = 1,$$
$$F_i = F_{i-2} + F_{i-1} \quad \text{for } i \geq 2.$$

To show that $\lambda x.F_x$ is primitive recursive, define $f = \lambda x.F_x$, $g = \lambda x.F_{x+1}$. Then

$$\begin{aligned} f(0) &= 0, & g(0) &= 1, \\ f(x+1) &= g(x), & g(x+1) &= f(x) + g(x). \end{aligned}$$

Now the assertion follows by theorem 2.6.

Exercise 2.8: Show that the following functions are primitive recursive without using $\mathcal{WL}$- or $\mathcal{AL}$-programs:

(i) [0] $\lambda x.x!$

(ii) [2] The function f defined by

$$f(x, y) = \begin{cases} \lfloor \log_x y \rfloor & \text{if } x \geq 2 \;\&\; y \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

We now turn to the last definition scheme in this section, namely *bounded* and *unbounded minimization*. The idea is to search for the smallest element satisfying some property.

Definition 2.7: Let $Q(x, \vec{y}_n)$ be a predicate. Then

(i) The function defined from Q by *unbounded minimization* is $\lambda \vec{y}_n.(\mu z)[Q(z, \vec{y}_n)]$, where

$$(\mu z)[Q(z, \vec{y}_n)] = \begin{cases} \text{the smallest } z \text{ such} & \\ \text{that } Q(z, \vec{y}_n) & \text{if such } z \text{ exists,} \\ \uparrow & \text{if no such } z \text{ exists.} \end{cases}$$

(ii) The function defined from Q by *bounded minimization* is $\lambda x\vec{y}_n.(\mu z < x)[Q(z, \vec{y}_n)]$, where

$$(\mu z < x)[Q(z, \vec{y}_n)] = \begin{cases} \text{the smallest } z < x & \\ \text{such that } Q(z, \vec{y}_n) & \text{if such } z \text{ exists,} \\ x & \text{if no such } z \text{ exists.} \end{cases}$$

The definition of the value $(\mu z < x)[Q(z, \vec{y}_n)]$ in the '$\neg\exists$-case,' where no $z < x$ satisfies $Q(z, \vec{y}_n)$, is largely arbitrary, and some textbooks set it to zero. Our definition, which is also applied in Tourlakis [25], for instance, has one advantage, though: the result $(\mu z < x)[Q(z, \vec{y}_n)] = x$ indicates that the $\neg\exists$-case does indeed hold.

Theorem 2.7: (i) If $Q(x, \vec{y}_n)$ is recursive, then $\lambda \vec{y}_n.(\mu z)[Q(z, \vec{y}_n)]$ is partial recursive.

(ii) If $Q(x, \vec{y}_n)$ is (primitive) recursive, then $\lambda x\vec{y}_n.(\mu z < x)[Q(z, \vec{y}_n)]$ is (primitive) recursive.

Proof: (i) The following program computes $\lambda \vec{y}_n.(\mu z)[Q(z, \vec{y}_n)]$ with respect to $Y_1, \ldots, Y_n, Z$:

$Z := 0;$
while $\neg Q(Z, \vec{Y}_n)$ **do**
    $Z := Z + 1$

If Q is recursive, then so is $\neg Q$, and the assertion follows.

(ii) A program computing $\lambda x\vec{y}_n.(\mu z < x)[Q(z, \vec{y}_n)]$ with respect to X, Y_1, …, Y_n, Z is like the one for (i), except that **while** $\neg Q(Z, \vec{Y}_n)$ has to be replaced by **while** $Z < X \,\&\, \neg Q(Z, \vec{Y}_n)$. Now the **while**-loop is bounded by X, hence theorem 2.3 applies for Q primitive recursive. □

Similarly as in parts (iv) and (v) of example 2.4 we may also apply bounded minimization of the form $\big(\mu z < h(\vec{x}_m, \vec{y}_n)\big)[\ldots]$. Using theorems 2.4–2.7, many number-theoretic functions are very easily shown to be primitive recursive (or partial recursive, if they are not total). We conclude this section by providing some more examples.

Example 2.5: The following functions and predicates will play an important rôle in the next section. In each case we provide a representation which shows that they are primitive recursive:

(i) The predicate (P) $\lambda xy.x|y$ (x divides y): $x|y \equiv y \bmod x = 0$.

(ii) The predicate (P) $\mathrm{Pr} = \lambda x.x$ is prime:

$$\mathrm{Pr}(x) \equiv x \geq 2 \,\&\, (\forall y < x)(y > 1 \Longrightarrow \neg y|x).$$

(iii) $\lambda x.p_x$, where (P) p_x is the x-th prime number, starting counting with 0. Thus $p_0 = 2$, $p_1 = 3$ etc. Here we need a result from number theory, namely a primitive recursive upper bound $f(x)$ for p_x. Given such a bound, $\lambda x.p_x$ can be defined by primitive recursion:

$$\begin{aligned} p_0 &= 2, \\ p_{x+1} &= \big(\mu z \leq f(x+1)\big)[\mathrm{Pr}(z) \,\&\, z > p_x]. \end{aligned}$$

We now give a proof (well-known from elementary texts on number theory) that $f(x) = 2^{2^x}$ is an upper bound for p_x. (There are much smaller upper bounds, but they are more difficult to prove.) The proof proceeds by induction of x. Clearly, $p_0 = f(0)$, $p_1 < f(1)$. Let $x \geq 2$, and assume $p_k \leq f(k)$ for all $k < x$. Consider $\tilde{p} = \prod_{i=0}^{x-1} p_i - 1$. Then $\tilde{p} > 1$, and none of p_0, …, p_{x-1} divides $\tilde{p}$, hence $\tilde{p}$ contains a prime divisor $> p_{x-1}$. Therefore $p_x \leq \tilde{p}$. Also, by induction hypothesis we have

$$\tilde{p} < \prod_{i=0}^{x-1} 2^{2^i} = 2^{\sum_{i=0}^{x-1} 2^i} = 2^{2^x - 1} < 2^{2^x}.$$

Combining these facts, it follows that $p_x < 2^{2^x}$. This completes the proof.

(iv) (P) $\lambda xy.\exp(x, y)$, where $\exp(x, y)$ is the exponent of p_x in the prime decomposition of y; since there is no prime decomposition of 0, define $\exp(x, 0) = 0$. We have

$$\exp(x, y) = (\mu z < y)[p_x^{z+1} \nmid y];$$

this definition also gives the right result for $y = 0$.

In exercises 2.9-2.12 you may use theorems 2.4–2.7 and functions which have been shown to be (primitive/partial) recursive earlier. However, you should not write new $\mathcal{WL}$- or $\mathcal{AL}$-programs.

Exercise 2.9: Show that the following functions are primitive recursive:

(i) [1] $\lambda x.\mathrm{npr}(x)$, where $\mathrm{npr}(x)$ is the number of primes $\leq x$. (This function is often denoted by π in textbooks on number theory, but in these lecture notes π will be used for something else.)

(ii) [2] $\lambda xy.\,\mathrm{lcm}(x, y)$, where $\mathrm{lcm}(x, y)$ is the least common multiple of x and y. (By definition $\mathrm{lcm}(x, 0) = \mathrm{lcm}(0, x) = 0$ for all x.)

Exercise 2.10: (i) [0] Let

$$f(x) = \begin{cases} \sqrt{x} & \text{if } x \text{ is a perfect square,} \\ \uparrow & \text{if } x \text{ is not a perfect square.} \end{cases}$$

Show that f is partial recursive.

(ii) [2] Let

$$f(n) = \begin{cases} \uparrow & \text{if } n \text{ is a Fermat-exponent,} \\ 0 & \text{if } n \text{ is a non-Fermat-exponent} \end{cases}$$

(see example 1.5). Show that f is partial recursive without using the solution for Fermat's problem by Wiles and Taylor.

Exercise 2.11: [2] Define

$$(\nu x < z)[Q(x, \vec{y}_n)] = \begin{cases} \text{the largest } x < z \text{ such} \\ \text{that } Q(x, \vec{y}_n) & \text{if such } x \text{ exists,} \\ 0 & \text{if no such } x \text{ exists.} \end{cases}$$

Show that if the predicate Q is (primitive) recursive, then so is the function $f = \lambda z\vec{y}_n.(\nu x < z)[Q(x, \vec{y}_n)]$.

Exercise 2.12: [4] Let $g_1, \ldots, g_4$ be (primitive/partial) recursive, and let f be defined by

$$\begin{aligned} f(0, \vec{y}_n) &= g_1(\vec{y}_n), \\ f(x, \vec{y}_n) &= g_2\big(x, \vec{y}_n, f(g_3(x), \vec{y}_n)\big) \quad \text{for } x > 0. \end{aligned}$$

Furthermore, assume that

$$(\forall x)\big(\exists k < g_4(x)\big)\big(g_3^k(x) = 0\big).$$

(If $g_4(x)\uparrow$, then it is not required that $g_3^k(x) = 0$ for any k, and $f(x, \vec{y}_n)$ is undefined if no such k exists.) Show that f is (primitive/partial) recursive.

2.3. Computability for other data types.

Intuitively, domains and ranges of 'computable' functions may consist of all kinds of mathematical objects. Therefore we wish to extend the property of being (primitive/ partial) recursive accordingly. As noted in section 1.1, this is done by encoding objects of various data types by natural numbers.

Let A be a countable set, regarded as a data type for variables. A *coding* (*function*) for A is a one-one function from A to ω. It is denoted by cd_A, and $\mathrm{cd}_A(a)$ is taken as the *code number* for a. The mapping cd_A should be simple, that is, $\mathrm{cd}_A(a)$ should be easily 'computable' from a. Also, it should be easily decidable if a given $x \in \omega$ is the code number of some $a \in A$; at least the set of all such code numbers has to be primitive recursive.

In what follows assume that cd_{A_i} has been defined for each of the sets A_i. Let $f: A_1 \times \cdots \times A_n \to A_{n+1}$. We say that $\tilde{f}: \omega^n \to \omega$ *corresponds to* f, iff for all $(a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$,

$$\tilde{f}\big(\mathrm{cd}_{A_1}(a_1), \ldots, \mathrm{cd}_{A_n}(a_n)\big) = \mathrm{cd}_{A_{n+1}}(f(a_1, \ldots, a_n)).$$

Such a function $\tilde{f}$ is denoted by f_ω; it is uniquely determined iff the mappings cd_{A_i} $(i = 1, \ldots, n)$ are onto. Similarly, if $B \subseteq A_1 \times \cdots \times A_n$, then the corresponding subset of ω^n is

$$B_\omega = \big\{ \big(\mathrm{cd}_{A_1}(a_1), \ldots, \mathrm{cd}_{A_n}(a_n)\big) \bigm| (a_1, \ldots, a_n) \in B \big\}.$$

Definition 2.8: A function $f: A_1 \times \cdots \times A_n \to A_{n+1}$ is (*primitive/partial*) recursive (with respect to codings $\mathrm{cd}_{A_1}, \ldots$), iff some corresponding function f_ω is (primitive/ partial) recursive; a set $B \subseteq A_1 \times \cdots \times A_n$ is (primitive) recursive or r.e. (with respect to $\mathrm{cd}_{A_1}, \ldots$), iff B_ω is (primitive) recursive or r.e. respectively.

Later on this definition will only be applied for a few specific data types A_i, and standard coding functions cd_{A_i} for them will be defined in the present section. The properties (primitive/partial) recursive and r.e. will always be meant with respect to these standard coding schemes, so they need not be specified.

We take the convention that the terms (primitive/partial) recursive or r.e. are used in this extended sense only when it is explicitly stated that they are applied to functions $f: A_1 \times \cdots \times A_n \to A_{n+1}$ or sets $B \subseteq A_1 \times \cdots \times A_n$. For instance, if we refer to the set of all (primitive) recursive or r.e. sets (without further specification), or if we state that there exists a (primitive/partial) recursive function with certain properties (without specifying its domain or range), then definitions 1.5, 1.6 (ii) and 1.7 apply.

Other properties $P(F)$ (mainly computability properties) of functions or sets F can also be extended to other data types, using an analogue of definition 2.8. In this case we say that P is *extended by coding.* However, such an extension should not be assumed automatically, for in some cases it works properly only if the coding functions are *onto.* Here is an example to illustrate this point.

In definition 6.10 a set $S \subset \omega$ will be defined to be *simple*, iff it is r.e. and $\overline{S}$ is infinite and contains no infinite r.e. subset. This means informally that $\overline{S}$ is somewhat sparse. Now let A be some data type other than ω, let cd_A be a corresponding coding function, and assume that $B = \overline{A_\omega}$ is infinite. (B is the set of all elements which are not code numbers of any $x \in A$.) Clearly, an r.e. subset $S \subseteq A$ should be regarded as simple iff $A - S$ is infinite and contains no infinite r.e. subset. However S_ω cannot be simple for any S, since B is an infinite subset of $\overline{S_\omega}$. Thus definition 2.8 cannot be applied with 'simple' in place of '(primitive/partial) recursive,' but this problem would disappear if cd_A were onto.

On the other hand, there also exist computability properties for which extension by coding does work, that is, the analogue of definition 2.8 can be applied for any data types. An example is the reducibility relation $\leq_T$ from definition 6.2.

For every data type A we have to specify a value 0_A, which plays the same rôle for A as 0 plays for ω. Our convention is that for each data type to be considered

0_A is the element which satisfies $\mathrm{cd}_A(0_A) = 0$; such an element will exist in each case (see exercise 2.24). However, in definitions and theorems concerning coding schemes in general it will not be assumed that $\mathrm{cd}_A(0_A) = 0$, but only that 0_A is defined. The element 0_A will be needed for the first time in theorem 2.10, and several times in section 2.4.

In the rest of this section we introduce standard coding schemes for certain specific data types, namely *tuples of natural numbers or other objects*, *strings over an alphabet*, *finite sets of natural number or other objects*, *integers* and *rational numbers*. The first three will be needed in later investigations and are therefore the most important. By contrast, integers and rational numbers are of little importance for the theory of recursive functions, though they are obvious examples for illustrating coding. Following the coding scheme for rational numbers, the notion of computable real number will be defined, which is an interesting topic by itself, but not one of the main topics in these lecture notes.

Except for rational numbers, all coding functions will be onto, which has the advantage described above. We impose this as an additional requirement on the coding schemes for tuples of natural numbers and for strings, even though it makes them slightly more complicated. For finite sets and for integers the simplest coding scheme already happens to be onto.

Tuples of natural numbers and of other objects. Two different coding schemes will be defined: one for tuples of constant, and one for tuples of variable length. We begin by defining coding functions for tuples of natural numbers of constant length $n \geq 1$; they will be denoted by ι_n. Consider first the case $n = 2$. The idea is simply to define $\iota_2(x, y)$ as the index of (x, y) in a straightforward enumeration of all elements of $\omega \times \omega$:

$$\begin{aligned} \iota_2(0, 0) &= 0, \\ \iota_2(0, 1) &= 1, \\ \iota_2(1, 0) &= 2, \\ \iota_2(0, 2) &= 3, \\ &\vdots \end{aligned}$$

Here the pairs (x, y) are ordered such that (x', y') comes before (x, y) iff either $x' + y' < x + y$, or $x' + y' = x + y$ and $x' < x$. $\iota_2(x, y)$ equals the number of pairs preceding (x, y) in this enumeration. For every z there are $z+1$ pairs (x', y') such that $x' + y' = z$ (namely $(0, z)$, $(1, z-1)$, $\ldots$, $(z, 0)$). Given (x, y), there are x pairs (x', y') preceding (x, y) such that $x' + y' = x + y$ (namely $(0, x + y)$, $\ldots$, $(x - 1, y + 1)$). Hence the index of (x, y) in our enumeration is

$$\sum_{z=0}^{x+y-1} (z + 1) + x = \frac{(x + y)(x + y + 1)}{2} + x.$$

This is the value assigned as $\iota_2(x, y)$; our discussion shows that ι_2 is a bijection from ω^2 to ω. Obviously, $\lambda x.\iota_2(x, y)$ and $\lambda x.\iota_2(y, x)$ are monotonously increasing for every y. Also, it follows from the results of section 2.2 that ι_2 is primitive recursive.

Definition 2.9: (i) The *pairing functions* ι_n, $n = 1, 2, \ldots,$ which serve as coding functions for ω^n, are defined recursively as follows:

$$\iota_1(x) = x, \quad \iota_2(x, y) = \frac{(x+y)(x+y+1)}{2} + x,$$
$$\iota_n(x_1, \ldots, x_n) = \iota_2(\iota_{n-1}(x_1, \ldots, x_{n-1}), x_n) \quad \text{for } n \geq 3.$$

(ii) More generally, if coding schemes cd_{A_i} have already been defined for countable sets A_i ($i = 1, \ldots, n$), then the coding scheme for $A = A_1 \times \cdots \times A_n$ is defined by

$$\mathrm{cd}_{A_1 \times \cdots \times A_n}\big((a_1, \ldots, a_n)\big) = \iota_n\big(\mathrm{cd}_{A_1}(a_1), \ldots, \mathrm{cd}_{A_n}(a_n)\big);$$

in this case $0_A = (0_{A_1}, \ldots, 0_{A_n})$. (This definition of 0_A is already covered by the general convention $\mathrm{cd}_A(0_A) = 0$, provided that the latter applies to all A_i, as it is the case in all our examples.)

Theorem 2.8: (i) ι_n is a bijection from ω^n to ω, primitive recursive and monotonously increasing in every argument.

(ii) Let (P) $\big(\kappa_1^n(x), \ldots, \kappa_n^n(x)\big)$ be the uniquely determined element of ω^n such that $\iota_n\big(\kappa_1^n(x), \ldots, \kappa_n^n(x)\big) = x$. Then the functions κ_i^n $(1 \leq i \leq n)$ are primitive recursive.

Proof: (i) For $n = 1$ the assertions are trivial, and for $n = 2$ they were already established above. For $n \geq 3$ they follow immediately by induction on n.

(ii) The simplest representation of κ_i^n which shows primitive recursiveness is

$$\kappa_i^n(x) = (\mu z_i \leq x)\big[(\exists z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_n \leq x)\big(\iota_n(z_1, \ldots, z_i, \ldots, z_n) = x\big)\big].$$

Note that $\iota_n(x_1, \ldots, x_n) \geq x_i$ for all $(x_1, \ldots, x_n)$ and for all i; this follows from the fact that ι_n is monotonously increasing in every argument. Hence $\kappa_i^n(x) \leq x$ for all i, n and x. □

Now let $f\colon A_1 \times \cdots \times A_n \to A_{n+1}$, as in definition 2.8 and before. In view of definition 2.9 (ii), f may be regarded in two different ways with respect to coding, leading to two different definitions of f_ω. Namely:

Option (α): f is regarded as a function of n arguments, and $\mathrm{cd}_{A_1}, \ldots, \mathrm{cd}_{A_n}, \mathrm{cd}_{A_{n+1}}$ are applied for coding. Then f_ω is again a function of n arguments.

Option (β): f is regarded as a function of one argument, and $\mathrm{cd}_{A_1 \times \cdots \times A_n}$ is applied for coding. Then of course f_ω is again a function of one argument.

Likewise, if $A \subseteq A_1 \times \cdots \times A_n$, then A_ω may be formed either by applying $\mathrm{cd}_{A_1}, \ldots,$ cd_{A_n}, or by applying $\mathrm{cd}_{A_1 \times \cdots \times A_n}$. Here, too, we refer to the two options as option (α) and option (β) respectively.

Suppose we wish to apply extension by coding to some property $P(F)$ originally defined for functions or sets F in $\mathcal{NF}$ or $\mathcal{NS}$ respectively. In order to show that the resulting extension does not depend on the choice between options (α) or (β), it suffices to show that the following condition is satisfied:

Definition 2.10: (N) A property $P(f)$ of functions $f \in \mathcal{NF}$ is *invariant under coding*, iff for every n and every $f\colon \omega^n \to \omega$ we have $P(f) \iff P(f')$, where $f'\bigl(\iota_n(\vec{x}_n)\bigr) = f(\vec{x}_n)$ for all $\vec{x}_n \in \omega^n$. A property $P(A)$ of sets $A \in \mathcal{NS}$ is *invariant under coding*, iff for every n and every $A \subseteq \omega^n$ we have $P(A) \iff P(A')$, where $A' = \iota_n(A)$.

Of course invariance under coding is defined in the obvious way for subsets $\mathcal{F}$ of $\mathcal{NF}$ or $\mathcal{NS}$, namely by the requirement that the predicate $F \in \mathcal{F}$ be invariant under coding.

Let P be invariant under coding, and apply the analogue of definition 2.8 to a property of functions $g\colon A_1 \times \cdots \times A_n \to A_{n+1}$ or sets $B \subseteq A_1 \times \cdots \times A_n$. Let f or A be the function g_ω or set B_ω obtained under option (α), f' or A' the one obtained under option (β). Then A and A' are related in the same way as in definition 2.10. The same is true for f and f' if f' is chosen appropriately for given f. Hence it makes no difference whether $P(g)$ ($P(B)$) is defined to be equivalent to $P(f)$ ($P(A)$) or to $P(f')$ ($P(A')$), and the extension by coding is unambiguous, as claimed before definition 2.10.

Theorem 2.9: The sets $\mathcal{PR}$, $\mathcal{R}$, $\mathcal{P}$, $\mathcal{PR}_*$, $\mathcal{R}_*$ and $\mathcal{E}$ are invariant under coding.

Proof: First let f and f' be as in definition 2.10. Then $f(\vec{x}_n) = f'\bigl(\iota_n(\vec{x}_n)\bigr)$ and $f'(x) = f\bigl(\kappa_1^n(x), \ldots, \kappa_n^n(x)\bigr)$, hence f is (primitive/partial) recursive iff f' is (primitive/partial) recursive. This shows that $\mathcal{PR}$, $\mathcal{R}$ and $\mathcal{P}$ are invariant under coding. Now let $\mathcal{S}$ be one of the sets $\mathcal{PR}_*$, $\mathcal{R}_*$ or $\mathcal{E}$. The claim that $A \in \mathcal{S} \iff \iota_n(A) \in \mathcal{S}$ for $A \subseteq \omega^n$ follows by applying the invariance of $\mathcal{PR}$, $\mathcal{R}$ and $\mathcal{P}$ to the characteristic function of A or to functions with domain A respectively; in the latter case we can conclude that some function with domain A is partial recursive iff some function with domain $\iota_n(A)$ is partial recursive. □

Exercise 2.13: [1] Using definition 2.8, the propery of being (primitive/partial) recursive is also defined for functions $f\colon A_1 \times \cdots \times A_n \to B_1 \times \cdots \times B_m$, where $m \geq 1$. Such a function can be represented in the form $f(\vec{x}_n) = \bigl(f_1(\vec{x}_n), \ldots, f_m(\vec{x}_n)\bigr)$. Show that f is (primitive/partial) recursive iff all f_i are (primitive/partial) recursive.

For convenience the definitions of some computability properties will be given only for functions $f \in \mathcal{NF}^{(1)}$ or sets $A \in \mathcal{NS}^{(1)}$ later in these lecture notes. Then it is understood that they extend to functions $f \in \mathcal{NF}^{(n)}$ or sets $A \in \mathcal{NS}^{(n)}$, by applying the analogue of definition 2.8 for the coding function ι_n; this is possible since ι_n is total. Equivalently, the extension can be obtained by stipulating that the property under consideration is invariant under coding. Examples are definitions 5.1, 6.2 and 6.10.

We now turn to coding tuples of variable length. The set of all tuples (of arbitrary length) of elements of a set A (including the tuple of length 0) is denoted by (P) A^*:

$$A^* = \bigcup_{i=0}^{\infty} A^i.$$

(Compare the paragraph after definition 2.3.) Consider first the case $A = \omega$. The coding scheme is based on the unique factorization of natural numbers $x > 0$: $x = \prod_{i=0}^{k-1} p_i^{x_i}$. (Recall that p_i is the i-th prime number.) However, this product cannot yet serve as the code number for $(x_0, \ldots, x_{k-1})$; a modification is necessary to guarantee that $\mathrm{cd}_{\omega^*}\colon \omega^* \to \omega$ is bijective.

Definition 2.11: (V) (i) The coding scheme for tuples of natural numbers of variable length is defined by

$$\text{(P) } \mathrm{cd}_{\omega^*}\big((x_0,\ldots,x_{n-1})\big) = \langle x_0,\ldots,x_{n-1}\rangle = \begin{cases} \big(\prod_{i<n-1} p_i^{x_i}\big)\cdot p_{n-1}^{x_{n-1}+1} - 1 & \text{for } n>0,\\ 0 & \text{for } n=0.\end{cases}$$

(ii) More generally, if a coding scheme cd_A for a countable set A has already been defined, then a coding scheme for A^* is defined by

$$\mathrm{cd}_{A^*}\big((a_0,\ldots,a_{n-1})\big) = \langle \mathrm{cd}_A(a_0),\ldots,\mathrm{cd}_A(a_{n-1})\rangle.$$

Exercise 2.14: [2] Explain the peculiar formula for $\langle x_0,\ldots,x_{n-1}\rangle$ in the case $n>0$: Why is 1 subtracted? Why is $x_{n-1}+1$ the exponent of p_{n-1}, but x_i the exponent of p_i for $i<n-1$? Also, prove that cd_{ω^*}, as defined, is indeed a bijection from ω^* to ω.

Theorem 2.10: Assume that cd_A and cd_B have been defined for data types A, B. The following functions and predicates are primitive recursive:

(i) The length function (P) $L^A\colon A^*\to\omega$, defined by

$$L^A\big((x_0,\ldots,x_{n-1})\big) = n.$$

(ii) The function (P) $\pi^A\colon A^*\times\omega\to A$, defined by

$$\pi^A\big((x_0,\ldots,x_{n-1}),i\big) = \begin{cases} x_i & \text{if } i<n,\\ 0_A & \text{if } i\geq n.\end{cases}$$

(iii) The function (P) $\mathrm{st}^A\colon A^*\times\omega\times A\to A^*$, defined by

$$\mathrm{st}^A\big((x_0,\ldots,x_{n-1}),i,a\big) = \begin{cases} (x_0,\ldots,x_{i-1},a,x_{i+1},\ldots,x_{n-1}) & \text{if } i<n,\\ (x_0,\ldots,x_{n-1},\underbrace{0_A,\ldots,0_A}_{i-n},a) & \text{if } i\geq n.\end{cases}$$

(iv) For every n the function $\lambda\vec{x}_n.(x_1,\ldots,x_n)$ from A^n to A^*.

(v) The function (P) $\mathrm{cat}^A\colon A^*\times A^*\to A^*$ defined by

$$\mathrm{cat}^A\big((x_0,\ldots,x_{n-1}),(y_0,\ldots,y_{m-1})\big) = (x_0,\ldots,x_{n-1},y_0,\ldots,y_{m-1}).$$

(vi) If $f\colon\omega\times B\to A$ is (primitive/partial) recursive, then so is the function $\tilde{f}\colon\omega\times B\to A^*$ defined by

$$\tilde{f}(z,y) = \big(f(0,y),\ldots,f(z-1,y)\big).$$

(If B is of type $B_1\times\cdots\times B_n$, then it does not matter whether f and $\tilde{f}$ are interpreted as functions of 2 or $n+1$ arguments, hence the case where these functions have more than two arguments is included in this part of the theorem. This remark also applies to other results, for instance exercises 2.19 and 2.20 below.)

Proof: (i) Using result and notation of exercise 2.11, we have

$$L^A_\omega(x) = (\nu z\leq x)[p_{z\dot{-}1}|x+1];$$

this also works for $x = 0$. Note that if $p_{z \dot{-} 1} | x+1$, then $z \le p_{z \dot{-} 1} - 1 \le x$.

(ii) This part and the next are shown without the assumption that $\mathrm{cd}_A(0_A) = 0$, so let $u = \mathrm{cd}_A(0_A)$. We have

$$\pi_\omega^A(x,y) = \begin{cases} \exp(y, x+1) & \text{if } y+1 < L_\omega^A(x), \\ \exp(y, x+1) \dot{-} 1 & \text{if } y+1 = L_\omega^A(x), \\ u & \text{if } y \ge L_\omega^A(x). \end{cases}$$

(iii) We have

$$\mathrm{st}_\omega^A(x,y,z) = \begin{cases} \lfloor (x+1)/p_y^{\exp(y,x+1)} \rfloor \cdot p_y^z - 1 & \text{if } y+1 < L_\omega^A(x), \\ \lfloor (x+1)/p_y^{\exp(y,x+1)} \rfloor \cdot p_y^{z+1} - 1 & \text{if } y+1 = L_\omega^A(x), \\ \lfloor (x+1)/p_{L_\omega^A(x) \dot{-} 1} \rfloor \cdot \prod_{i=0}^{(y \dot{-} L_\omega^A(x))-1} p_{i+L_\omega^A(x)}^u \cdot p_y^{z+1} - 1 & \text{if } y \ge L_\omega^A(x) \;\&\; x > 0, \\ \prod_{i=0}^{y-1} p_i^u \cdot p_y^{z+1} - 1 & \text{if } x = 0. \end{cases}$$

(iv) This follows immediately from theorem 2.4; the primes $p_0, \ldots, p_{n-1}$ serve as constants in the expression for $\langle \mathrm{cd}_A(x_1), \ldots, \mathrm{cd}_A(x_n) \rangle$.

(v) We define a function f such that

$$f(k, \langle x_0, \ldots, x_{n-1} \rangle, \langle y_0, \ldots, y_{m-1} \rangle) = \langle x_0, \ldots, x_{n-1}, y_0, \ldots, y_{k-1} \rangle$$

for $k \le m$; the value of $f(k,x,y)$ is irrelevant if $k > L_\omega^A(y)$. Such a function is obtained by primitive recursion:

$$\begin{aligned} f(0,x,y) &= x, \\ f(k+1,x,y) &= \mathrm{st}_\omega^A\big(f(k,x,y), L_\omega^A(x)+k, \pi_\omega^A(y,k)\big). \end{aligned}$$

Then

$$\mathrm{cat}_\omega^A(x,y) = f\big(L_\omega^A(y), x, y\big).$$

(vi) We have

$$\begin{aligned} \tilde{f}_\omega(0,y) &= \langle \rangle, \\ \tilde{f}_\omega(z+1,y) &= \mathrm{st}_\omega^A\big(\tilde{f}_\omega(z,y), z, \mathrm{cd}_A(f(z,y))\big). \end{aligned}$$

□

Parts (i)–(iii) are the crucial properties of the coding scheme cd_{A^*}; any other scheme satisfying them would be just as suitable. It is easy to see that parts (iv)–(vi) can be derived from (i)–(iii). For parts (v) and (vi) this follows directly from the proof, and obviously the functions $\lambda \vec{x}_n.(x_1, \ldots, x_n)$ could also be expressed in terms of st.

Exercise 2.15: [2] Show that the function $f\colon A^* \to A^*$ satisfying

$$f\big((x_0, \ldots, x_{n-1})\big) = (x_{n-1}, \ldots, x_0)$$

is primitive recursive.

Alphabets, strings and languages. The next data type to be encoded are strings over an alphabet, and we take the occasion to introduce the basic notions of formal languages theory:

Definition 2.12: (i) An *alphabet* is a finite, non-empty set of symbols. In the following parts of this definition let Σ be an alphabet.

(ii) A *word* (*string*) over Σ is a sequence of elements of Σ, represented by juxtaposing them.

(iii) The *length* of a string w, denoted by $|w|$, is the number of symbols of which w consists.

(iv) The *empty word* (*string*), denoted by ϵ, is the word consisting of zero symbols.

(v) Σ^* denotes the set of all words, Σ^+ the set of all non-empty words over Σ.

(vi) A *(formal) language* over Σ is a subset of Σ^*.

(vii) The *concatenation* of w, $x \in \Sigma^*$, denoted by wx, is the sequence consisting of the symbols of w followed by those of x. Similarly, the concatenation $w_1 \dots w_k$ of k strings $w_1, \dots, w_k$ is defined in the obvious way, and w^n is the concatenation of n copies of w. In this notation a string consisting of a single symbol $a \in \Sigma$ may be represented by a itself, so that expressions like ax, a^n etc. have the obvious meaning.

(viii) Let w, $x \in \Sigma^*$. Then w is a prefix (suffix, substring) of x, iff there exist strings y (and z) such that $x = wy$ ($x = yw$, $x = ywz$ respectively).

The coding function cd_{Σ^*} for strings over Σ will be based on an enumeration $(\sigma_0, \dots, \sigma_{m-1})$ of Σ; it is understood that this enumeration is fixed along with Σ. Similarly as for ω^2 it is obtained from a straightforward listing of Σ^*, where strings are ordered first by length, then alphabetically:

string	code
ϵ	0
σ_0	1
$\dots$	$\dots$
σ_{m-1}	m
$\sigma_0\sigma_0$	$m+1$
$\dots$	$\dots$

The code number $\mathrm{cd}_{\Sigma^*}(w)$ of $w = \sigma_{i_0} \dots \sigma_{i_{l-1}}$ can now be determined as follows: First, there are m^j strings of length j for every j, hence the number of strings of length $< l$ is

$$\sum_{j=0}^{l-1} m^j = \sum_{j=0}^{l-1} m^{l-1-j} = \begin{cases} \frac{m^l-1}{m-1} & \text{if } m > 1, \\ l & \text{if } m = 1. \end{cases} \tag{1}$$

The strings of length exactly l are orderd like the m-ary numbers of l digits (including leading 0's), hence the rank of w among these strings (starting counting at 0) is

$$(i_0 i_1 \dots i_{l-1})_m = \sum_{j=0}^{l-1} i_j m^{l-1-j}. \tag{2}$$

The sum of the numbers from (1) and (2) is the desired index of w. Hence we define:

Definition 2.13: The coding function for Σ^* with respect to an enumeration $(\sigma_0, \ldots, \sigma_{m-1})$ for Σ is defined by

$$\mathrm{cd}_{\Sigma^*}(\sigma_{i_0} \ldots \sigma_{i_{l-1}}) = \sum_{j=0}^{l-1} (i_j + 1) m^{l-1-j}.$$

The preceding discussion shows:

Theorem 2.11: cd_{Σ^*} is a bijection from Σ^* to ω. $\square$

The following theorem parallels to some extent theorem 2.10.

Theorem 2.12: Let the alphabet Σ and its enumeration $(\sigma_0, \ldots, \sigma_{m-1})$ be fixed.

(i) The length function (P) $L^\Sigma \colon \Sigma^* \to \omega$, defined by

$$L^\Sigma(w) = |w|$$

is primitive recursive.

(ii) The function (P) $\pi^\Sigma \colon \Sigma^* \times \omega \to \omega$, defined by

$$\pi^\Sigma(\sigma_{i_0} \ldots \sigma_{i_{l-1}}, x) = \begin{cases} i_x & \text{if } x < l, \\ 0 & \text{if } x \geq l \end{cases}$$

is primitive recursive.

(iii) Assume that a coding scheme cd_A has been defined for data type A. If $f \colon \omega \times A \to \omega$ is (primitive/partial) recursive and $f(x, y) < m$ for all x, y, then the function $\check{f} \colon \omega \times A \to \Sigma^*$, defined by

$$\check{f}(z, y) = \sigma_{f(0,y)} \ldots \sigma_{f(z-1,y)}$$

is also (primitive/partial) recursive.

Also, the following functions and predicates are primitive recursive:

(iv) The function (P) $\mathrm{cat}^\Sigma \colon \Sigma^* \times \Sigma^* \to \Sigma^*$ defined by $\mathrm{cat}^\Sigma(x, y) = xy$.

(v) The predicate (P) $\mathrm{pref}^\Sigma(x, y)$ on Σ^* which holds iff x is a prefix of y.

(vi) The predicate (P) $\mathrm{suff}^\Sigma(x, y)$ on Σ^* which holds iff x is a suffix of y.

(vii) The predicate (P) $\mathrm{substr}^\Sigma(x, y)$ on Σ^* which holds iff x is a substring of y.

(viii) The function (P) $\mathrm{subst}^\Sigma \colon (\Sigma^*)^3 \to \Sigma^*$ such that $\mathrm{subst}^\Sigma(x, y, z)$ is the result of replacing the first occurrence of x in z by y. (If x is not a substring of z, then put $\mathrm{subst}^\Sigma(x, y, z) = z$.)

Proof: (i) We have

$$L^\Sigma_\omega(x) = \begin{cases} (\mu z \leq x)[x < \frac{m^{z+1} \dot{-} 1}{m \dot{-} 1}] & \text{if } m > 1, \\ x & \text{if } m = 1. \end{cases}$$

Note that this is not an application of theorem 2.5, since m is fixed; it only has to be shown that L^Σ is primitive recursive both for $m > 1$ and for $m = 1$.

(ii) From $z = \mathrm{cd}_{\Sigma^*}(\sigma_{i_0} \ldots \sigma_{i_{l-1}})$ one obtains the number $(i_0 \ldots i_{l-1})_m$ by subtracting $(m^l - 1)/(m-1)$, as noted above; then one obtains i_x by first dividing by m^{l-x-1}, which eliminates the trailing $l - x - 1$ digits, and then taking the congruence value mod m. Thus

$$(\pi^\Sigma)_\omega(z,x) = \begin{cases} \left\lfloor \left(z \dot{-} \frac{m^{L^\Sigma_\omega(z)} \dot{-} 1}{m \dot{-} 1} \right) \Big/ m^{L^\Sigma_\omega(z) \dot{-} (x+1)} \right\rfloor \bmod m & \text{if } m > 1, \\ 0 & \text{if } m = 1. \end{cases}$$

This definition also gives the right result, namely 0, if $x \geq L^\Sigma_\omega(z)$.

(iii) From the formula for cd^Σ in definition 2.13 it can be seen that

$$\mathrm{cd}_{\Sigma^*}(x\sigma_k) = m\,\mathrm{cd}_{\Sigma^*}(x) + k + 1.$$

Hence we have

$$\begin{aligned} \check{f}_\omega(0,y) &= 0, \\ \check{f}_\omega(z+1,y) &= m\check{f}_\omega(z,y) + f_\omega(z,y) + 1. \end{aligned}$$

Parts (iv)–(viii) are dealt with in the following exercise. □

Exercise 2.16: [2] Complete the proof of theorem 2.12.

Note that the notation L^Σ, π^Σ, cat^Σ from theorem 2.12 is consistent with the notation L^A, π^A, cat^A from theorem 2.10, unless a distinction is made between the terms *tuple* and *sequence*. Of course, the corresponding functions on natural numbers are not the same, due the different coding schemes in definitions 2.11 and 2.13. The functions L^A_ω, π^A_ω etc. constructed in the proof of theorem 2.10 are independent of the data type A, but if the coding $\mathrm{cd}_A\colon A \to \omega$ is not onto, then other functions could also serve as L^A_ω etc. Similarly, the functions L^Σ_ω etc. of theorem 2.12 depend only on $|\Sigma|$, not on Σ itself. No analogue to the function st^A has been defined for strings, since none will be needed; conversely, analogues of pref^Σ, suff^Σ etc. are of little importance for tuples of natural numbers.

From now on superscripts, as well as a possible subscript ω, will be omitted from the functions L, π, etc. and predicates pref, suff etc., unless they are necessary to avoid confusion.

Exercise 2.17: [3] (i) Show that for $f\colon (\Sigma^*)^n \to \Sigma^*$ the property of f being (primitive/partial) recursive is independent of the enumeration of Σ.

(ii) Show that for $A \subseteq (\Sigma^*)^n$ the property of A being (primitive) recursive (r.e.) is independent of the enumeration of Σ.

Finite sets. For every set A we denote the set of all finite subsets of A by (P) $A^\#$; hence the set of all finite subsets of $A^\#$ is denoted by $A^{\#\#}$ etc.

Definition 2.14: (i) The code number of a finite set $F \in \omega^\#$ is the number

$$\mathrm{cd}_{\omega^\#}(F) = \sum_{z \in F} 2^z;$$

it is called the *canonical index* of F. If x is the canonical index of F, then F is denoted by D_x.

(ii) More generally, if A is a countable set for which a coding scheme cd_A has already been defined, then the coding function for $A^\#$ is defined by

$$\mathrm{cd}_{A^\#}(F) = \sum_{z \in F} 2^{\mathrm{cd}_A(z)}.$$

The claim that every $x \in \omega$ is the canonical index of a uniquely determined $F \in \omega^\#$ follows from the fact that every natural number has a unique representation as a sum of powers of 2, which is reflected by its binary representation. Similarly, in part (ii) of definition 2.14, $\mathrm{cd}_{A^\#}$ is a bijection from $A^\#$ to ω, provided that cd_A is a bijection from A to ω.

Finite sets and canonical indices will be dealt with again in chapter 5. They are needed time and again in recursive function theory, and these lecture notes also contain a few applications. See, for instance, theorem 5.18 and exercise 6.9.

Exercise 2.18: Show that the following functions and predicates are primitive recursive:

(i) [2] The predicate $\lambda xX.x \in X$ on $A \times A^\#$.

(ii) [0] The function $\lambda x.\{x\}$ from A to $A^\#$.

(iii) [2] The functions $\lambda XY.X \cup Y$, $\lambda XY.X \cap Y$ and $\lambda XY.X - Y$ from $A^\# \times A^\#$ to $A^\#$.

(iv) [2] The function $\lambda X.|X|$ from $A^\#$ to ω.

(v) [1] The function $\lambda X.\max(X)$ from $A^\#$ to A, where $\max(X)$ is the element of X whose code number is maximal; $\max(\emptyset)$ is defined to be 0_A.

(vi) [0] The predicate $\lambda XY.X \subseteq Y$ on $A^\# \times A^\#$.

(vii) [2] The function $\lambda X.2^X$ from $A^\#$ to $A^{\#\#}$.

Exercise 2.19: Let A, B be data types for which cd_A and cd_B have been defined.

(i) [2] Show that if $f\colon A \times B \to \omega$ is (primitive/partial) recursive, then so are the functions $\lambda Xy.\sum_{x\in X} f(x,y)$ and $\lambda Xy.\prod_{x\in X} f(x,y)$ from $A^\# \times B$ to ω.

(ii) [0] Show that if the predicate P on $A \times B$ is (primitive) recursive, then so are the predicates $\lambda Xy.(\exists x \in X)\big(P(x,y)\big)$ and $\lambda Xy.(\forall x \in X)\big(P(x,y)\big)$ on $A^\# \times B$.

(iii) [2] Show that if the predicate P on $\omega \times B$ is (primitive) recursive, then so is the function $g\colon \omega^\# \times B \to \omega$ defined by

$$g(X,y) = \begin{cases} \text{the smallest } z \in X \text{ such that } P(z,y) & \text{if such a } z \text{ exists,} \\ 0 & \text{if no such } z \text{ exists.} \end{cases}$$

(iv) [1] Show that if $f\colon A \to B$ is (primitive/partial) recursive, then so is the function $g\colon A^\# \to B^\#$ defined by

$$g(X) = \{\, f(x) \mid x \in X \,\}.$$

The coding scheme for $A^\#$ can also be used to encode finite *functions*. Denote the set of all finite functions from A to B by $\mathcal{F}(A,B)$. This includes the case where f is a function of n arguments from $A_1, \ldots, A_n$, since we may choose $A = A_1 \times \cdots \times A_n$.

As usual we assume that coding schemes cd_A, cd_B have been defined. Then a coding scheme for $\mathcal{F}(A, B)$ can be defined by

$$\mathrm{cd}_{\mathcal{F}(A,B)}(F) = \mathrm{cd}_{(A\times B)^{\#}}\big(\mathrm{gr}(F)\big).$$

If z is the code of a function $F \in \mathcal{F}(A, B)$, then

$$D_z = \{\, \mathrm{cd}_{A\times B}\big((x, y)\big) \mid F(x) = y \,\}.$$

A subset of $A \times B$ is the graph of a function iff it contains no two different elements (x, y), (x', y') with $x = x'$. Hence z is the code number of some element of $\mathcal{F}(A, B)$ iff

$$(\forall x, x' \in D_z)\big(x \neq x' \Longrightarrow \kappa_1^2(x) \neq \kappa_1^2(x')\big).$$

By exercise 2.19 (ii) it follows that the set of all such code numbers is primitive recursive.

Of course every operation f on finite functions which corresponds to a (primitive/partial) recursive set operation g for their graphs is again (primitive/partial) recursive, since $f_\omega = g_\omega$. Some more operations are considered in the following exercise.

Exercise 2.20: Show that the following functions and predicates are primitive recursive:

(i) [1] The function $\lambda F.\,\mathrm{dom}(F)$ from $\mathcal{F}(A, B)$ to $A^{\#}$.

(ii) [1] The function $\lambda F.\,\mathrm{ran}(F)$ from $\mathcal{F}(A, B)$ to $B^{\#}$.

(iii) [2] The function $\lambda Fx.F(x)$ from $\mathcal{F}(A, B) \times A$ to B.

(iv) [1] The predicates 'F is one-one,' 'F is onto' and 'F is bijective' on $\mathcal{F}(A, B)$.

(v) [2] The function $\lambda FD.F(D)$ from $\mathcal{F}(A, B) \times A^{\#}$ to $B^{\#}$.

(vi) [2] The function $\lambda FD.F^{-1}(D)$ from $\mathcal{F}(A, B) \times B^{\#}$ to $A^{\#}$.

(vii) [2] The function $\lambda FG.FG$ from $\mathcal{F}(B, C) \times \mathcal{F}(A, B)$ to $\mathcal{F}(A, C)$.

(viii) [2] Some function $f\colon \mathcal{F}(A, B) \to \mathcal{F}(B, A)$ which satisfies $f(F) = F^{-1}$ whenever F is one-one.

Integers, rational and real numbers. Following the convention applied in many textbooks, we denote the set of of integers, rational numbers, and the set of real numbers by $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ respectively. True, this does not fit well to the notation ω for the set of natural numbers, for which many books use the symbol $\mathbb{N}$.

The coding schemes for *integers* and *rational numbers* are defined as follows:

$$\mathrm{cd}_{\mathbb{Z}}(x) = \begin{cases} 2x - 1 & \text{for } x > 0, \\ -2x & \text{for } x \leq 0, \end{cases}$$
$$\mathrm{cd}_{\mathbb{Q}}(u) = \iota_2\big(\mathrm{cd}_{\mathbb{Z}}(x), y\big), \quad \text{where } u = x/y,\ x \in \mathbb{Z},\ y \in \omega - \{0\},\ \gcd(x, y) = 1.$$

Note that every rational number u has a unique representation of the form x/y with the conditions specified.

The following exercise shows that the ordinary operations and relations on integers and rational functions are primitive recursive.

Exercise 2.21: (i) [2] Show that the function $\lambda x.|x|$ from $\mathbb{Z}$ to ω and the functions $\lambda xy.x+y$, $\lambda xy.x-y$ and $\lambda xy.xy$, as well as predicates $\lambda xy.x<y$ and $\lambda xy.x\le y$ on $\mathbb{Z}$ are primitive recursive.

(ii) [2] Show the same for the corresponding functions and predicates on $\mathbb{Q}$, and include also the function

$$f(x,y)=\begin{cases} x/y & \text{if } y\neq 0,\\ 0 & \text{if } y=0.\end{cases}$$

Also, show that $\{\,\mathrm{cd}_{\mathbb{Q}}(u)\mid u\in\mathbb{Q}\,\}$ is primitive recursive.

We now consider the concept of *computable real number*. The property of a real number α of being computable can be defined by each of the following conditions (R1)–(R3). Condition (R3) says that any given decimal digit of α can be effectively computed. The function f_α is defined as follows: Represent α in the form $\alpha=\lfloor\alpha\rfloor+\alpha_0$, where $0\le\alpha_0<1$. Then $f_\alpha(0)=\mathrm{cd}_{\mathbb{Z}}(\lfloor\alpha\rfloor)$, and for $x>0$ $f_\alpha(x)$ is the x-th digit after the decimal point of the decimal representation of α_0. Here one has to take into account that certain real numbers have two decimal representations, one ending in 0's, one ending in 9's, for instance: $0.500\ldots=0.499\ldots$. This ambiguity is avoided by stipulating that the second representation is never used. The three conditions are as follows:

(R1) The predicate $\lambda u.u<\alpha$ on $\mathbb{Q}$ is recursive.

(R2) The predicate $\lambda xy.y>0\ \&\ x/y<|\alpha|$ on ω^2 is recursive.

(R3) The function f_α is recursive.

Exercise 2.22: (i) [1] Show that each of the conditions (R1)–(R3) is satisfied if $\alpha\in\mathbb{Q}$.

(ii) [2] Show that each of the conditions (R1)–(R3) holds for α iff it holds for $-\alpha$. (This part is still quite straightforward, but a rigorous proof of the assertion for (R3) requires some work.)

(iii) [4] Show that the conditions (R1)–(R3) are equivalent. (The tricky part is to show that (R3) implies (R1) or (R2).)

Definition 2.15: A real number α is *computable* (*recursive*), iff it satisfies the equivalent conditions (R1)–(R3).

Exercise 2.23: Show that the predicate $\lambda xy.y>0\ \&\ x/y<|\alpha|$ is in fact primitive recursive, if α is one of the following:

(i) [0] $\sqrt{2}$;

(ii) [4] Any *algebraic* real number, that is, any real number α satisfying a relation of the form $p(\alpha)=0$, where p is a nonzero polynomial function with integer coefficients;

(iii) [4] Euler's number $e=\sum_{i=0}^{\infty}1/i!$.

Exercise 2.24: [0] What is the element 0_A for each of the data types A considered in this section?

2.4. Application of coding to recursion and programming.

The results of section 2.3 can be used to extend once again the class of programs and definition schemes for constructing (primitive/partial) recursive functions. We begin by considering programs with data types other than ω. Of course, the programming language is still $\mathcal{AL}(\mathcal{F})$, but now $\mathcal{F}$ may contain functions $f\colon A_1\times\cdots\times A_n\to A_{n+1}$

and/or predicates $Q \subseteq A_1 \times \cdots \times A_n$, where each A_i is either ω or another data type (which then has to be included in $\mathcal{F}$, too). The following syntactical rules apply for such programs:

1. If a variable X is of a type $T \neq \omega$, then X has to be declared in the form

$$X : T$$

The variable declarations precede the rest of the program and are separated from each other and the rest by semicolons. Similarly, the types of parameters and local variables have to be specified in procedure declarations. Several variables of the same type may be included in a single declaration. For instance, the beginning of a program may be as follows:

X, Y: ω^*;
A, B: $\mathbb{Z}^{\#}$;
procedure $Pr(X{:}\,\mathbb{Z};$ **var** $U, V{:}\,\omega^2)$;
local $A{:}\,\omega^{\#}$;
⋮

Undeclared variables, parameters etc. are still of type ω by default.

2. Assignments, conditional statements etc. are as before, but the expressions involved may be of any data type, and there must not be any type clashes. For instance, if X and Y are of type ω, U is of type $\mathbb{Z}$ and f is a function from $\omega \times \omega$ to $\mathbb{Z}$, then the assignment $U := f(X, Y)$ is ok, but $U := f\big(f(X, Y), X\big)$ or $X := f(X, Y)$ is not.

3. In a **for**-statement **for** $I := E_1$ **to** E_2 **do** S the (common) data type of the control variable I and the expressions E_1, E_2 must be ω or $\mathbb{Z}$, otherwise there would be no well-defined sequence of values from E_1 to E_2.

4. Data types of the form A^* are interpreted as *arrays*, and the common syntax for arrays may be used. For instance, if U is of type ω^*, then $U[U[1]] := U[2]+U[3]$ is a valid assignment statement. The subscript must always be of type ω. Multiple subscripts are abbreviations for iterated ones, for instance, $X[i_1, i_2, i_3]$ actually means $X[i_1][i_2][i_3]$. Subscripted variables like $U[t]$ are regarded as variables, hence they can occur at the left side of assignment statements and as variable parameters in procedure calls. (Of course they can also occur as value parameters, since the latter may be arbitrary expressions.)

The *semantics* of $\mathcal{AL}(\mathcal{F})$ is defined in a straightforward manner, using the following conventions. If a variable X of data type A is not an input variable, then it is initialized to 0_A. Of course, this rule applies to global variables (to be initialized at the beginning of the program execution) as well as local variables declared in a procedure Pr (to be initialized when Pr is called). 0_A is also the value of $Y[t]$ if Y is of type A^* and the value of the expression t is $\geq L(Y)$. One last detail concerns the effect of statements of the form $X[t] := E$. If the values of $L(X)$ and t are l and m prior to this assignment, then the value of $L(X)$ is $\max(l, m+1)$ afterwards. This is in accordance with an application of the function st to X, as defined in theorem 2.10 (iii).

We also generalize the representation of functions by programs by allowing several variables to be specified as output variables. Let $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ be variables of a program P of data types $A_1, \ldots, A_n, B_1, \ldots, B_m$. It is natural to associate with P

a function $f: A_1 \times \cdots \times A_n \to B_1 \times \cdots \times B_m$ defined as follows: if $f(\vec{x}_n) = \vec{y}_m$, then the computation of P for input $\vec{x}_n$ terminates with Y_i having value y_i, but if $f(\vec{x}_n)\uparrow$, then this computation does not terminate. We call this the function computed by P with respect to input variables $X_1, \ldots, X_n$ and output variables $Y_1, \ldots, Y_m$. Of course, this convention does not increase the power of $\mathcal{AL}$, since $Y_1, \ldots, Y_m$ could be replaced by a single output variable of type $B_1 \times \cdots \times B_m$. But occasionally it makes writing $\mathcal{AL}$-programs more convenient, as in the proof of theorem 3.1 (vi). When we say that P computes f with respect to variables $U_1, \ldots, U_k$ (without saying explicitly which are input- and which are output variables), then the original convention applies, that is, only U_k is an output variable.

Theorem 2.13: Theorems 2.1 and 2.3 remain valid if $\mathcal{F}$ contains data types other than ω and/or if more than one variable is specified as output variable.

Proof: Assume that P computes f with respect to input variables $X_1, \ldots, X_n$ and output variables $Y_1, \ldots, Y_m$. Then P computes functions f_i with respect to $X_1, \ldots, X_n, Y_i$, where $f(\vec{x}_n) = \big(f_1(\vec{x}_n), \ldots, f_m(\vec{x}_n)\big)$. By exercise 2.13 it follows that the assertion about f follows from corresponding assertions for $f_1, \ldots, f_m$. This means that the case $m > 1$ reduces to the case $m = 1$.

Now let f, P be as in theorem 2.1, except that the data types are no longer restricted to ω. We show how to convert P into a program P' of $\mathcal{AL}(\mathcal{F}')$ which satisfies the following conditions:

(a) $\mathcal{F}'$ contains no data types, and only partial recursive functions and recursive predicates. Moreover, if $\mathcal{F}$ contains only primitive recursive functions and predicates, then the same is true for $\mathcal{F}'$.

(b) If P satisfies the conditions of theorem 2.3, then so does P'. Note that this includes the case where P satisfies the conditions of theorem 2.1 (ii).

(c) P' computes f_ω.

For the following proof it is assumed that iterated array subscripts are *not* replaced by multiple ones. The conversion proceeds as follows:

1. Eliminate procedure calls and procedure declarations. This is done as in the proof of theorem 2.1, except that local variables of type A have to be initialized to 0_A rather than 0. (This step has to be carried out before steps 3 and 4, since neither of these two steps works properly for subscripted variables as *variable parameters*.)

2. Replace every **for**-statement **for** $I := E_1$ **to** E_2 **do** S with I of type $\mathbb{Z}$ similarly as in step 2 in the proof of theorem 2.1. The new variables V_1 and V_2 are now of type $\mathbb{Z}$, and modified substraction may be replaced by ordinary subtraction. It has to be shown that the **while**-loops thus introduced are bounded by an expression of the form $f(Z_1, \ldots, Z_n)$ with f primitive recursive; this is one of the requirements of exercise 2.25 below.

3. Replace every expression of the form $X[t]$ with X of type A^* by $\pi^A(X, t)$, provided that $X[t]$ with possible additional subscripts *is not the left side of an assignment statement*. For example, $X[t]$ may occur at the right side of assignment statements, inside subscripts on the left side of assignment statements, or inside the Boolean conditions for **while**- and conditional statements.

4. Replace every assignment of the form $X[t] := E$, X of type A^*, by $X := \mathrm{st}^A(X, t, E)$. ($X$ may itself be subscripted, in which case this step has to be repeated

several times before subscripts are eliminated.)

5. Finally, change the type of every variable to ω, that is, eliminate all variable declarations, and replace every function f by f_ω. □

Exercise 2.25: [2] Complete the proof of theorem 2.13 by showing that the program P' constructed really satisfies conditions (a)–(c) specified in that proof.

Notation: We write $X[i]$ (where X is of type A^*) not only in programs. If x is of type ω, then we write $x.i$ for $\pi^\omega_\omega(x,i)$, $x.i.j$ for $\pi^\omega_\omega\big(\pi^\omega_\omega(x,i),j\big)$ etc.

Example 2.6: Let R be a countable ring, and assume that a coding scheme cd_R for ring elements has been defined such that $\mathrm{cd}_R(0)=0$, and that addition, subtraction and multiplication of ring elements are primitive recursive. Every polynomial $a_nx^n+\cdots+a_0$ ($a_n\neq 0$) over R can be represented by the tuple $(a_0,\ldots,a_n)$, and the zero polynomial is represented by the empty tuple, of course. This leads to a coding scheme for the polynomial ring $R[x]$:

$$\mathrm{cd}_{R[x]}(a_nx^n+\cdots+a_0)=\langle \mathrm{cd}_R(a_0),\ldots,\mathrm{cd}_R(a_n)\rangle.$$

Not every tuple corresponds to a polynomial, since trailing 0's are excluded.

In order to show that a function $f\colon (R[x])^n\to R[x]$ is (primitive/partial) recursive, it suffices to show that this is the case for a corresponding function $\hat f\colon (R^*)^n\to R^*$, for then $\hat f_\omega$ can also serve as f_ω. We now show that addition, subtraction and multiplication of polynomials are primitive recursive. An auxiliary function tr is needed which eliminates trailing 0's from a tuple, that is

$$\begin{aligned}\mathrm{tr}\big((a_0,\ldots,a_{n-1},0,\ldots,0)\big)&=(a_0,\ldots,a_{n-1})\quad\text{for } a_{n-1}\neq 0,\\ \mathrm{tr}\big((0,\ldots,0)\big)&=().\end{aligned}$$

This function is primitive recursive by theorem 2.3, since the following program, whose **while**-loop is bounded by $L(X)$, computes tr_ω with respect to X and X:

$Z := L(X) \dot- 1$;
while $Z>0\ \&\ X.Z>0$ **do**
  begin $X := (X+1)*p_{Z\dot-1}/p_Z \dot- 1$; $Z := Z\dot- 1$ **end**;
if $X=1$ **then** $X:=0$

Here the statement $X:=(X+1)*p_{Z\dot-1}/p_Z\dot-1$ eliminates a trailing 0 from the tuple with code X, provided that $L_\omega(X)\dot-1=Z\geq 1$, but this does not work for $X=\langle 0\rangle$ (see definition 2.11).

Now the following programs compute functions from $R^*\times R^*$ to R^* corresponding to polynomial addition, subtraction and multiplication with respect to X, Y, Z:

Addition and subtraction:

X, Y, Z: R^*;
for $I:=0$ **to** $\max\big(L(X),L(Y)\big)\dot-1$ **do**
  $Z[I]:=X[I]\pm Y[I]$;
$Z:=\mathrm{tr}(Z)$

Multiplication:

$X, Y, Z\colon R^*$;
for $I := L(X) + L(Y) \dot{-} 2$ **do**
$\quad Z[I] := \sum_{j=0}^{I} X[j] \cdot Y[I-j]$

Note that the subscripted variables $X[I]$, $Y[I]$ and $Z[I]$ are of type R, and addition etc. were assumed to be primitive recursive for such elements. Also, the value of $\sum_{j=0}^{I} X[j] \cdot Y[I-j]$ is a primitive recursive function of the values of X, Y and I, since addition for ring elements is another example of an operation to which example 2.4 (iv) carries over. Thus theorem 2.13 applies, and we are done.

In general, results like those in section 2.2 can always be extended by allowing arbitrary data types for the variables. These extensions are justified by replacing all variables of data types other than ω by their code numbers, as it was done in the last step of the proof of theorem 2.13. For theorems 2.4 and 2.5 this generalization requires no comment. The same is true for theorems 2.6 and 2.7, if the first arguments of the functions f_i or of the predicate Q respectively are of type ω. Assume now that they are of some other type A. Then definition 2.6 must be modified by replacing 0 by $0_A = \mathrm{cd}_A^{-1}(0)$ and $x+1$ by $\sigma'(x)$, where σ' satisfies $\mathrm{cd}_A(\sigma'(x)) = \mathrm{cd}_A(x)+1$; this works only if cd_A is onto. Similarly, in definition 2.7 the word 'smallest' refers to the linear ordering $\leq'$ on A which is induced by the ordinary ordering $\leq$ on ω and the mapping cd_A, that is,

$$x \leq' y \equiv \mathrm{cd}_A(x) \leq \mathrm{cd}_A(y).$$

With these modifications theorems 2.6 and 2.7 carry over. Other results of this type can be generalized similarly.

Example 2.7: Let $\Sigma = \{\sigma_0, \ldots, \sigma_{m-1}\}$, and let $h\colon (\Sigma^*)^{n+1} \to \Sigma^*$ be defined from functions $f\colon (\Sigma^*)^n \to \omega$ and $g_0, \ldots, g_{m-1}\colon (\Sigma^*)^{n+2} \to \omega$ by

$$\begin{aligned} h(\epsilon, \vec{y}_n) &= f(\vec{y}_n), \\ h(x\sigma_i, \vec{y}_n) &= g_i\big(x, \vec{y}_n, h(x, \vec{y}_n)\big). \end{aligned}$$

This is a more natural recursion scheme for functions on strings than primitive recursion. We show that if the functions f and g_i are (primitive/partial) recursive, then so is h. In fact, this is an application of exercise 2.12, with Σ^* in place of ω. Denote the functions g_1–g_4 of that exercise by $\hat{g}_1$–$\hat{g}_4$. Clearly, $\hat{g}_1 = f$. For $x \neq \epsilon$ let $\hat{g}_3(x)$ be the string obtained from x by deleting the last character. It was noted in the proof of theorem 2.12 (iii) that $\mathrm{cd}_{\Sigma^*}(x\sigma_k) = m\mathrm{cd}_{\Sigma^*}(x) + k + 1$; this implies that $(\hat{g}_3)_\omega(x) = \lfloor (x \dot{-} 1)/m \rfloor$. (The value of $\hat{g}_3(\epsilon)$ is irrelevant, of couse.) The function L^Σ can be chosen as $\hat{g}_4$. Finally, $\hat{g}_2$ is defined as follows:

$$\hat{g}_2(x, \vec{y}_n, z) = \begin{cases} g_0(x, \vec{y}_n, z) & \text{if } \pi^\Sigma\big(x, L^\Sigma(x) \dot{-} 1\big) = 0, \\ \quad\vdots & \\ g_{m-1}(x, \vec{y}_n, z) & \text{if } \pi^\Sigma\big(x, L^\Sigma(x) \dot{-} 1\big) = m-1. \end{cases}$$

Up to this point the results of this section displayed only one new feature in comparison to earlier ones, namely the consideration of data types other than ω. This is not the case for the following two theorems, the last ones of this chapter. Both use arrays in an essential way, theorem 2.14 already in its statement, theorem 2.15 in its proof.

In ordinary mathematics a function $f: \omega \times A \to B$ may be defined by induction, such that the definition of $f(x, y)$ uses the values $f(z, y)$ for all $z < x$. The following definition is based on this idea. Since no functions with a variable number of arguments are admitted, all the values $f(z, y)$, $z < x$, have to be collected in a tuple:

Definition 2.16: A function $f: \omega \times A \to B$ is defined from $g: \omega \times A \times B^* \to B$ by *course-of-values recursion*, iff for all x, y

$$f(x, y) = g\big(x, y, (f(0, y), \ldots, f(x-1, y))\big).$$

Theorem 2.14: If f is defined from g by course-of-values recursion and g is (primitive/ partial) recursive, then f is (primitive/partial) recursive.

Proof: As in theorem 2.10 (vi) let $\tilde{f}(z, y) = \big(f(0, y), \ldots, f(z-1, y)\big)$. Then

$$\begin{aligned} \tilde{f}(0, y) &= (), \\ \tilde{f}(z+1, y) &= \mathrm{st}^B\big(\tilde{f}(z, y), z, g(z, y, \tilde{f}(z, y))\big). \end{aligned}$$

Hence $\tilde{f}$ is (primitive/partial) recursive. Finally, $f(x, y) = \pi^B\big(\tilde{f}(x+1, y), x\big)$. □

Of course, theorem 2.14, like theorem 2.6, carries over to the case where recursion is applied to some other argument of f than the first. Recall that there may be more than two arguments, see the remark after the statement of theorem 2.10.

Example 2.8: Let $f: \omega \to \mathbb{Q}$ be defined by

$$\begin{aligned} &f(0) = 0, \quad f(1) = 2, \\ &f(x) = x + 1 + \frac{2}{x} \sum_{i=0}^{x-1} f(i) \quad \text{for } x \geq 2. \end{aligned}$$

(In fact, $f(x)$ is the average number of comparisons of the QUICKSORT-algorithm for arrays of x elements.) Let h be the function $\lambda x.x$ from ω to $\mathbb{Q}$. Then $h_\omega = \lambda x.\iota_2(2x \dot{-} 1, 1)$, hence h is primitive recursive. Now the function f is obtained by course-of-values recursion from the function $g: \omega \times \mathbb{Q}^* \to \mathbb{Q}$ defined as follows:

$$g(x, z) = \begin{cases} 0 & \text{if } x = 0, \\ 2 & \text{if } x = 1, \\ h(x) + 1 + \frac{2}{h(x)} \sum_{i=0}^{x \dot{-} 1} \pi^{\mathbb{Q}}(z, i) & \text{if } x \geq 2. \end{cases}$$

Using exercise 2.21 (ii) we can conclude that g is primitive recursive. Note that taking sums of rational numbers is another operation for which example 2.4 (iv) applies. It follows that f is primitive recursive, too.

Exercise 2.26: [2] Formulate and proof a theorem which combines theorems 2.6 and 2.14 in a natural way.

The most general method for defining functions recursively is by recursive procedures. Every definition scheme considered so far can easily be represented that way. The following exercise provides an illustration:

Exercise 2.27: [2] Let f be as in example 2.8. Design a procedure Pr such that the procedure call $Pr(X, Z)$ has the effect of assigning to Z the value $f(X)$ *without using any of the functions like* $\pi^{\mathbb{Q}}$ *introduced in theorem 2.10.* Use the function h of example 2.8 to avoid type clashes between ω and $\mathbb{Q}$.

Finally, we extend theorems 2.1 and 2.13 to the case where P contains recursive procedures. In this case the function computed is no longer necessarily primitive recursive, even if the conditions of theorem 2.1 (ii) or theorem 2.3 are otherwise satisfied.

Theorem 2.15: If the functions in $\mathcal{F}$ are partial recursive and the predicates in $\mathcal{F}$ recursive, then every function computed by an $\mathcal{AL}(\mathcal{F})$-program P is partial recursive.

Proof: As usual we describe an algorithm for converting P into an equivalent program P' without procedures. This algorithm only serves the theoretical purpose of proving the theorem and does not represent a practical method which might be applied by real compilers. The method of theorem 2.1 does no longer work. Assume, for instance, that Pr calls itself. If a call of Pr from the main program is replaced by the body of Pr (with formal parameters replaced by actual ones), then another call of Pr is left (namely that from Pr to itself). Hence a completely new idea is needed. This time the conversion does not proceed step by step; instead P' is constructed directly.

Let $Pr_0, \ldots, Pr_p$ be a list of all procedures of P, where Pr_0 is P itself. At every stage during an execution of P every ongoing execution of Pr_i is called an *activation* of Pr_i. For example, at the beginning there is only one activation (of Pr_0). If Pr_0 calls (an instance of) Pr_2, which in turn calls Pr_1, which in turn calls Pr_2, then after the latter call there are one activation of Pr_0, one of Pr_1 and two of Pr_2. When the execution of P is finished, no procedure activation is left.

Let S_i be the body of Pr_i. The body of the new program P', that is, the portion of P' after the declarations of variables of types other than ω, will of the form

$$T_0;\ \textbf{goto}\ END;\ T_1;\ \ldots;\ T_p;\ END:,$$

where T_i is obtained by modifying S_i as described below. The basic idea is as follows: If S_i contains a call of Pr_j, then T_i will instead contain a jump to the first statement of T_j, so that T_j will be executed subsequently. This execution of T_j will be equivalent to that of Pr_j for the current parameters. Of course, at the end of Pr_j there must be statements which effect a jump back to the right place within T_i.

The main difficulty involved is that one identifier of S_i does not always correspond to the same storage location. There are two reasons for this. For one thing, assume that during an execution of P there are several activations of Pr_i. If X_i is a local variable of Pr_i of type A, then P', while simulating P, must save the value of X_i for *every* activation of Pr_i. Hence instead of X_i, P' will use a variable U_i of type A^*. Secondly, assume that X_i is a formal variable parameter of Pr_i. When Pr is executed, then X_i is linked to the corresponding actual parameter, which may differ from call to call; it may also be an array component in some cases.

This problem is resolved by introducing for every formal parameter X_i of P a variable H_i which indicates for every procedure activation, which variable or array component of P' X_i corresponds to. For formal value parameters and local variables this purpose is served by another new variable D, which indicates the current number of procedure activations besides that of Pr_0. P' contains yet another array C which plays the following rôle: If at some stage of an execution of P there are procedure activations Pr_{i_0}, …, Pr_{i_m}, then $C[j]$ indicates where P' has to jump after the simulation of Pr_{i_j} is finished. All these variables have to be initialized when P' simulates a procedure call of P. The same is true for the variables of P' corresponding to formal value parameters and local variables of the procedure called, as in the proof of theorem 2.1.

We now come to the details of the construction. Let X_0, …, X_{m_1-1} be the *global* variables of P, X_{m_1}, …, X_{m_2-1} those identifiers which serve as formal value parameters in some procedure, X_{m_2}, …, X_{m_3-1} those which are *local* variables, and X_{m_3}, …, X_{m_4-1} those which serve as formal variable parameters. P' contains variables U_0, …, U_{m_3-1} corresponding to X_0, …, X_{m_3-1} (but none corresponding to X_{m_3}, …, X_{m_4-1}). If X_i is of type A, then U_i is of type A for $i \leq m_1 - 1$, or of type A^* for $m_1 \leq i \leq m_3 - 1$. In addition, it has already been stated that P' contains variables C, D and H_i ($m_3 \leq i \leq m_4 - 1$); their types are ω^*, ω and ω^{**} respectively.

In order to construct T_i from S_i, we start by replacing every identifier X_j by U_j for $i \leq m_1 - 1$, or by $U_j[D]$ for $m_1 \leq i \leq m_3 - 1$. Formal variable parameters X_j ($j \geq m_3$) are handled by conditional statements, as illustrated by the following example. Suppose S_i contains the following assignment I: $X_j := X_j \cup \{0\}$, where X_j is a formal variable parameter of type $\omega^{\#}$. Assume further that $m_1 = 4$, $m_3 = 15$, that X_2, X_5 and X_{12} have types $\omega^{\#*}$, $\omega^{\#}$ and $\omega^{\#**}$ respectively, whereas the other X_k with $k \leq 14$ are not of one of the types $\omega^{\#}$, $\omega^{\#*}$, $\omega^{\#**}$ etc. Then I has to be replaced by

> **case** $H_j[D, 0]$ **of**
> > 2: $U_2[H_j[D, 1]] := U_2[H_j[D, 1]] \cup \{0\}$;
> > 5: $U_5[D] := U_5[D] \cup \{0\}$;
> > 12: $U_{12}[D, H_j[D, 1], H_j[D, 2]] := U_{12}[D, H_j[D, 1], H_j[D, 2]] \cup \{0\}$;
> > **else**

We see how $H_j[D]$ encodes the linkage between the formal variable parameter X_j and a variable or array component of P': $H_j[D, 0]$ is the index of this variable, and $H_j[D, 1]$, $H_j[D, 2]$ etc. indicate which component or component of a component etc. has to be extracted. If a statement of S_i contains several variable parameters X_{i_0}, X_{i_1}, …, then in the corresponding statement of T_i more cases are needed to account for every possible combination of values $H_{i_0}[D]$, $H_{i_1}[D]$ etc. Conditionals, **while**-statements etc., where formal parameters can occur in various expressions, are handled in the same fashion.

Let I_0, I_1, …, I_r be the procedure calls in the various procedures Pr_i. Every call I_j in Pr_i will be replaced by a sequence of statements R_j in T_i, as described shortly. The last instruction of R_j will be the empty statement M_j :. This is where P' has to jump after simulating the execution of the procedure Pr_k called by I_j. Accordingly, the following statements are attached at the end of T_k:

> $D := D \dot{-} 1$;
> **case** $C[D]$ **of**
> > 0: **goto** M_0;
> > ⋮
> > r: **goto** M_r;
> > **else**

In some cases it would also be possible to omit the empty statement M_j : and to attach M_j to the statement immediately after R_j (or to use the label which that statement already had). But this does not work if I_j is the last statement of the body of a **while**-statement, of the **then**- or **else**-part of a conditional or something similar, or of Pr_i itself.

A label also has to be attached to the first statement of T_i, if that statement was unlabelled before; denote this label by L_i.

We are now ready for a description of R_j. We assume that the procedure call I_j occurs within Pr_i and has the form $Pr_k(E_1, \ldots, E_m, V_1, \ldots, V_l)$, where $E_1, \ldots, E_m$ are expressions, $V_1, \ldots, V_l$ are variables with or without subscripts, and the heading of the procedure Pr_k is **procedure** $Pr_k(X_{i_1}, \ldots, X_{i_m}; \textbf{var } X_{j_1}, \ldots, X_{j_l})$. As noted above, a number of variables have to be initialized for the simulation of Pr_k, and accordingly R_j contains the following instructions:

(a) The statements $D := D + 1$; $C[D] := j$.

(b) Statements $H_q[D] := e_q$ which assign the right values to $H_q[D]$ for $q = m_3, \ldots, m_4 - 1$. If X_q is not a formal variable parameter of Pr_k, then X_q remains linked to whatever it was linked in Pr_i, hence e_q is the expression $H_q[D \dot{-} 1]$. Otherwise $q = j_t$ for some t, and X_q is linked to V_t. V_t is of the form $X_{t'}[d_1, \ldots, d_s]$ for expressions $d_1, \ldots, d_s$. Now put $e_q = (t', d_1, \ldots, d_s)$ if $t' \leq m_3 - 1$, or $e_q = \mathrm{cat}(H_{t'}[D \dot{-} 1], (d_1, \ldots, d_s))$ if $t' \geq m_3$.

(c) The statements $X_{i_s} := E_{i_s}$ $(s = 1, \ldots, m)$ and $X_{q'} := 0_{A_{q'}}$ for all local variables $X_{q'}$ belonging to P_k, where $A_{q'}$ is the data type of $X_{q'}$.

(d) The statements **goto** L_k; M_j :.

In the last step of the construction of T_i we have to deal with **goto**-statements of S_i: **goto** L. If the statement of P labelled L occurs within S_i, too, then this **goto**-statement is left unchanged in T_i. Otherwise the jump may terminate several procedure activations, so the new value of D has to be computed correctly. Let $Pr_{i'}$ be the procedure of P in which the statement with label L occurs. $Pr_{i'}$ is an ancestor of Pr_i, otherwise **goto** L would be a jump *into* some procedure, which is not allowed. Hence there exists some activation of $Pr_{i'}$, since descendants of $Pr_{i'}$ can only be called via $Pr_{i'}$. Let S be the set of all t such that the procedure call I_t occurs within $Pr_{i'}$, or, equivalently, the empty statement M_t : occurs within T_i. If the current procedure activations are $Pr_{i_0}, \ldots, Pr_{i_m}$, then $i_s = i'$ iff $C[s+1] \in S$. Hence **goto** L has to be replaced by the following statements, where S is a constant:

$D := D \dot{-} 1$;
while $C[D+1] \notin S$ **do**
    $D := D \dot{-} 1$;
goto L

This completes the construction of T_i and of P'. □

Exercise 2.28: [2] Do exercise 2.7 (ii) again, using theorem 2.15.

3. Universal functions, diagonalization and basic theory of r.e. sets.

Up to now we were concerned primarily with one objective: providing new tools for proving that certain functions are (primitive/partial) recursive. These tools are of course fundamental, and many of them will be needed over and over, but those results which are really typical for the theory of recursive functions have not been touched yet. Such results are to a large extend negative: Certain functions are *not* recursive, certain recursive functions are *not* primitive recursive etc. It is clear that such assertions cannot be shown by the methods used so far. Instead, the proofs usually proceed by one of these two fundamental methods: *diagonalization* and *reduction*. The first of these methods will be applied first in the proof of theorem 3.7, the second will be dealt with in section 5.1.

The following section 3.1 contains those theorems which are fundamental for the further development of the theory, whereas sections 3.2 and 3.3 contain additional basic results in which those theorems are applied. The chapter concludes with a discussion of ChT (section 3.4).

3.1. Universal functions.

In order to establish the existence of nonrecursive functions and sets, a simple cardinality argument is already sufficient. It is based on the following observations.

Firstly, every $\mathcal{WL}$-program may be regarded as a string over some fixed alphabet Σ. Σ contains capital letters, digits, the blank, the symbols $>$, $+$ and $\dot{-}$, and the tokens **while**, **do** and **end**. Theorem 2.11 implies that Σ^* is countable, so we can conclude:

(I) The set of all $\mathcal{WL}$-programs is countable.

Secondly, fix an input variable X and an output variable Y. Every $\mathcal{WL}$-program P computes a unique partial recursive function with respect to X and Y, and for every one-place partial recursive function f there exists a $\mathcal{WL}$-program which computes f with respect to X and Y. Hence $\mathcal{P}^{(1)}$ cannot have larger cardinality than that of the set of $\mathcal{WL}$-programs. (In this argument X and Y were fixed, since otherwise the function computed by P would not be unique.) Also, recursive sets $A \subseteq \omega$ are in one-one correspondence with one-place recursive characteristic functions, and the latter are contained in $\mathcal{P}^{(1)}$. Thus we have

(II) $\mathcal{P}^{(1)}$ is countable, hence the same follows for $\mathcal{R}^{(1)}$ and $\mathcal{R}^{(1)}_*$.

On the other hand, a well-known theorem by Cantor states:

(III) For every set A, $|A| < |2^A|$.

Applying this to $A = \omega$, it follows that 2^ω is uncountable. Now (II) and (III) yield:

(IV) There exist nonrecursive sets $A \subseteq \omega$.

Of course, if A is nonrecursive, then c_A is an example of a function $f\colon \omega \to \omega$ which is not recursive, (and hence not partial recursive, since f is total).

From the point of view of computability theory, somewhat stronger results than (IV) are of interest. Such results state the existence of functions or sets which lie outside some computability class but inside a larger one, as indicated at the beginning of this chapter. Hence we wish to check if such results can be obtained by analysing Cantor's proof of (III). Specialized to the case $A = \omega$, this proof runs as follows. Let A_0, A_1, $\ldots$ be a (countable) list of subsets of ω; we have to show that there exists some $A \subseteq \omega$ which is not included in this list. Define $A = \{ x \mid x \notin A_x \}$. Then A cannot coincide with any of the sets A_n, since $n \in A \iff n \notin A_n$. Thus A does indeed satisfy the required condition. This is a typical application of the diagonalization method, which will be explained in more detail in section 3.2.

Let us try to apply this argument in order to construct a recursive set which is not primitive recursive. In this case A_0, A_1, $\ldots$ should be a list of $\mathcal{PR}_*^{(1)}$. It follows as before that $A = \{ x \mid x \notin A_x \} \notin \mathcal{PR}_*$, but we have to make sure that $A \in \mathcal{R}_*$. Applied to characteristic functions, the latter requirement takes the following form:

(∗) If $f_i = c_{A_i}$, then $f = \lambda x.1 \dot{-} f_x(x)$ has to be recursive.

It will be seen that (∗) is indeed satisfied if the list A_0, $\ldots$ is chosen appropriately; in this way we will obtain theorem 3.7 in section 3.2. In fact, a stronger assertion holds: the function $\lambda nx.f_n(x)$ is recursive. It is the *universal function* corresponding to the sequence $(f_0, f_1, \ldots)$. This consideration clearly shows the relevance of universal functions for recursive function theory. A formal definition follows:

Definition 3.1: (i) Let $\mathcal{F}$ be a countable subset of $\mathcal{NF}^{(n)}$. An *enumeration* of $\mathcal{F}$ is a sequence of functions f_0, f_1, $\ldots$ such that $\mathcal{F} = \{ f_u \mid u \in \omega \}$.

(ii) The corresponding *universal function* for $\mathcal{F}$ is the function $g = \lambda u\vec{x}_n.f_u(\vec{x}_n)$. (Thus, if g is given in the first place, then g is a universal function for $\mathcal{F}$ iff the sequence of functions f_0, f_1, $\ldots$ defined by $f_u = \lambda \vec{x}_n.g(u, \vec{x}_n)$ is an enumeration of $\mathcal{F}$; this is the enumeration corresponding to g.)

(iii) If $\mathcal{S}$ is a countable subset of $\mathcal{NS}^{(n)}$, then an enumeration of $\mathcal{S}$ is a sequence of sets A_0, A_1, $\ldots$ such that $\mathcal{S} = \{ A_i \mid i \in \omega \}$.

Using the notation from this definition, the universal function g may be regarded as a single object which encodes each of the functions f_i. Indeed, f_i can be extracted from it in the form $f_i = \lambda x_1 \ldots x_n.g(i, \vec{x}_n)$.

We now turn to the main task of this section: the construction of one fixed enumeration $\phi_0^{(n)}$, $\phi_1^{(n)}$, $\ldots$for $\mathcal{P}^{(n)}$ for which the corresponding universal function $\psi^{(n)}$ is partial recursive (theorem 3.2) and satisfies some additional fundamental properties (theorems 3.3, 3.4 and 3.6). These results are not very hard to prove, but they require a certain amount of work and make good use of the results from chapter 2. Using theorem 3.2, it will be easy to establish assertions like (∗) above, where $\{f_0, f_1, \ldots\}$ is a proper subset of $\mathcal{P}$. See the proof of theorem 3.7.

The enumeration $\phi_0^{(n)}$, $\phi_1^{(n)}$, $\ldots$ is obtained in a straightforward way by introducing code numbers for $\mathcal{WL}$-programs. Namely, if program P has code number u, then $\phi_u^{(n)}$ is the n-place function computed by P with respect to a tuple of variables that will be fixed once and forever. We note that the universal function ψ is essentially what an interpreter $\mathcal{I}$ for $\mathcal{WL}$ computes. The only difference is that $\mathcal{I}$ takes a program P, rather than its code number u, as its first input. Given P and input data $\vec{x}_n$ which serve as arguments for ϕ_u, $\mathcal{I}$ will yield $\phi_u(\vec{x}_n)$ as output.

For the rest of this section let X_0, X_1, $\ldots$ be a fixed enumeration of the set of all identifiers. In our construction of the universal function neither the programs themselves nor their code numbers are convenient objects for computation. Instead we use *list representations* $\hat{I}$, $\hat{P}$ and $\hat{\mathcal{C}}$ for $\mathcal{WL}$-statements I, $\mathcal{WL}$-programs P and configurations $\mathcal{C}$

of types ω^*, ω^{**} and $\mathcal{CF} = \omega^{**} \times \omega \times \omega^*$ respectively. For every type of statement I, $\hat{I}$ is given in the following table.

statement I	list representation $\hat{I}$
$X_i := X_i + 1$	$(0, i)$
$X_i := X_i \dot{-} 1$	$(1, i)$
while $X_i > 0$ **do**	$(2, i)$
end	(3)

Thus if I is an instruction, then $\hat{I}[0]$ indicates what type of instruction it is, and $\hat{I}[1]$ indicates the variable involved (except for **end**-statements).

The list representation of $\mathcal{WL}$-programs is obvious: if $P = I_0; \ldots; I_{l-1}$, then

$$\hat{P} = (\hat{I}_0, \ldots, \hat{I}_{l-1}).$$

With such programs we also associate a code number (Gödel number):

$$\mathrm{cd}_{\mathcal{P}}(P) = \mathrm{cd}_{\omega^{**}}(\hat{P}).$$

Finally, a configuration $\mathcal{C} = (C, V)$ of a URM $U = (R, P)$ is represented by a triple $\hat{\mathcal{C}} = (\hat{P}, C, X)$ of the type $\mathcal{CF}$ specified above such that $X[i] = V(X_i)$ for all $X_i \in R$. Note that $\hat{\mathcal{C}}$ includes information about the program, but not about the set of registers of U; the latter plays no rôle in what follows. $\hat{\mathcal{C}}$ is not unique, since trailing 0's may be attached to or deleted from the tuple X.

Definition 3.2: (V) Let $n \geq 0$ be given. The functions $\phi_u^{(n)}$ $(u \in \omega)$ are defined as follows: If $u = \mathrm{cd}_{\mathcal{P}}(P)$ for a $\mathcal{WL}$-program P, then $\phi_u^{(n)}$ is the function $f\colon \omega^n \to \omega$ computed by P with respect to $X_1, \ldots, X_n, X_{n+1}$; but if u is not the code number of a $\mathcal{WL}$-program, then $\phi_u^{(n)}(\vec{x}_n) = 0$ for all $\vec{x}_n$. Also, $\psi^{(n)}$ is the corresponding universal function, that is,

$$\psi^{(n)}(u, \vec{x}_n) = \phi_u^{(n)}(\vec{x}_n).$$

Obviously, $\phi_u^{(n)}$ $(u = 0, 1, \ldots)$ is indeed an enumerations of $\mathcal{P}^{(n)}$, as claimed. The upper index (n) in $\phi_u^{(n)}$ or $\psi^{(n)}$ will be omitted whenever $n = 1$.

Theorem 3.1: The following functions and predicates are primitive recursive:

(i) A function $\mathrm{pl}\colon \omega \to \omega^{**}$ such that $\mathrm{pl}(u) = \hat{P}$ if $u = \mathrm{cd}_{\mathcal{P}}(P)$, and $\mathrm{pl}(u)$ does not equal $\hat{P}$ for any $\mathcal{WL}$-program P if u does not equal $\mathrm{cd}_{\mathcal{P}}(P)$ for any P.

(ii) The predicate $\mathrm{wl}(U)$ on ω^{**} which is true iff $U = \hat{P}$ for some $\mathcal{WL}$-program P.

(iii) The predicate $\mathrm{swl}(U)$ on ω^{**} which is true iff $U = \hat{P}$ for some *special* $\mathcal{WL}$-program P.

(iv) For every $n \geq 0$ a function $\mathrm{init}^{(n)}\colon \omega^{**} \times \omega^n \to \mathcal{CF}$ which satisfies the following: If $U = \hat{P}$ for some $\mathcal{WL}$-program P, then $\mathrm{init}^{(n)}(U, \vec{x}_n) = \hat{\mathcal{C}}$, where $\mathcal{C}$ is the initial configuration of the computation of P for input $\vec{x}_n$.

(v) A predicate halt on $\mathcal{CF}$ such that $\mathrm{halt}(Z)$ holds iff $Z = \hat{\mathcal{C}}$ for some halting configuration $\mathcal{C}$, or if Z does not equal $\hat{\mathcal{C}}$ for any configuration $\mathcal{C}$.

(vi) A function $\mathrm{next}: \mathcal{CF} \to \mathcal{CF}$ which satisfies $\mathrm{next}(\hat{\mathcal{C}}) = \hat{\mathcal{C}}'$ whenever $\mathcal{C}$ and $\mathcal{C}'$ are configurations of some URM and $\mathcal{C}$ yields $\mathcal{C}'$ in one step.

(vii) For every $n \geq 0$ a function $\mathrm{res}^{(n)}$ which extracts the result of a computation from its final configuration. Thus if the computation of $f(\vec{x}_n)$ on some URM terminates in the configuration $\mathcal{C}$, then $f(\vec{x}_n) = \mathrm{res}^{(n)}(\hat{\mathcal{C}})$.

Proof: For some parts of the proof we need auxiliary functions f_1–f_5 which are not of sufficient interest to be included in the statement of the theorem. The functions f_1, $f_2: \omega^{**} \times \omega \to \omega$ and $f_3: \omega^{**} \times \omega \to \mathbb{Z}$ correspond to the functions $\#_1$, $\#_2$ and d from exercise 1.1, except that $\hat{P}$ occurs as an additional argument. Thus $f_1(\hat{P}, x)$ is the number of **while**-, $f_2(\hat{P}, x)$ the number of **end**-statements among $I_0, \ldots, I_{x-1}$, and $f_3(\hat{P}, x)$ is their difference. These functions are all defined by an obvious primitive recursion, for instance

$$f_1(U, 0) = 0,$$
$$f_1(U, x+1) = \begin{cases} f_2(U, x) + 1 & \text{if } U[x, 0] = 2, \\ f_2(U, x) & \text{if } U[x, 0] \neq 2. \end{cases}$$

See remark 2.4. and the discussion between examples 2.6 and 2.7.

The functions f_4 and $f_5: \omega^{**} \times \omega \to \omega$ are defined such that if $P = I_0; \ldots; I_{l-1}$ is a $\mathcal{WL}$-program and I_x is a **while**- (**end**-) statement, then $I_{f_4(\hat{P}, x)}$ ($I_{f_5(\hat{P}, x)}$) is the matching **end**- (**while**-) statement. The (easy) proof that these functions are primitive recursive is left for exercise 3.1.

Now we turn to the individual assertions of the theorem.

(i) Using the notation of theorem 2.10 (vi) we have

$$\mathrm{pl}(u) = \tilde{f}\big(L(u), u\big),$$

where

$$f(z, u) = \begin{cases} (u.z.0, u.z.1) & \text{if } (\exists x \leq 2)(\exists y < u)(u.z = \langle x, y \rangle), \\ (3) & \text{if } u.z = \langle 3 \rangle, \\ () & \text{otherwise.} \end{cases}$$

Note that the case $u.z.0 \geq 4$ occurs only if u is *not* the code number of some $\mathcal{WL}$-program, and in this case the value of $\mathrm{pl}(u)$ is irrelevant.

(ii) Using exercise 1.1 (i), we have

$$\begin{aligned} \mathrm{wl}(U) \equiv \big(\forall x < L(U)\big)\big((U[x, 0] \leq 2 \,\&\, L(U[x]) = 2) \vee U[x] = (3)\big) \\ \&\, \big(\forall y < L(U)\big)\big(f_3(U, y) \geq 0\big) \,\&\, f_3\big(U, L(U)\big) = 0. \end{aligned}$$

(iii) This part is again left for exercise 3.1.

(iv) We have

$$\mathrm{init}^{(n)}(U, \vec{x}_n) = \big(U, 0, (0, x_1, \ldots, x_n)\big).$$

To see that this works, note the following. Let $A = (0, x_1, \ldots, x_n))$, and let $\mathcal{C} = (C, V)$ be the initial configuration of the computation of P for input $\vec{x}_n$. Then for $i > n$ we have $A[i] = 0$ (by the conventions of section 2.4) and $V(X_i) = 0$ (since all variables except for the input variables $X_1, \ldots, X_n$ are initialized to 0). Thus we have $(\forall i)\big(A[i] = V(X_i)\big)$, as required.

(v) Put

$$\mathrm{halt}\big((U, C, X)\big) \equiv C \geq L(U) \vee \neg\mathrm{wl}(U).$$

(vi) This time primitive recursiveness is shown by providing a program P_0 which computes next with respect to input variables U, C, X and output variables U, C, X, see theorem 2.13. Thus if the initial values of U, C and X represent $\hat{\mathcal{C}}$, then after execution of P_0 their values represent $\hat{\mathcal{C}}' = \mathrm{next}(\mathcal{C})$. The cases considered correspond to definition 1.4 (ii), but they are organized somewhat differently, so as to shorten the code.

$U\colon \omega^{**}$; $X\colon \omega^{*}$;
$I := U[C, 0]$; $J := U[C, 1]$;
if $I = 0$ **then** $X[J] := X[J] + 1$
  else if $I = 1$ **then** $X[J] := X[J] \dot{-} 1$;
if $I = 2\ \&\ X[J] = 0$ **then** $C := f_4(U, C) + 1$
  else if $I = 3$ **then** $C := f_5(U, C)$
  else $C := C + 1$

(vii) We have

$$\mathrm{res}^{(n)}\big((U, C, X)\big) = X[n + 1].$$

□

Exercise 3.1: [1] Complete the proof of theorem 3.1 by showing that the following are primitive recursive:

(i) The functions f_4 and f_5. *Hint:* Use exercise 1.1 (ii).
(ii) The predicate swl.

Theorem 3.2: The functions $\psi^{(n)}$ $(n \in \omega)$ are partial recursive.

Proof: The following $\mathcal{AL}$-program computes $\psi^{(n)}$ with respect to X_0, X_1, …, X_n, X_{n+1}:

$A\colon \mathcal{CF}$;
$A := \mathrm{init}^{(n)}(\mathrm{pl}(X_0), X_1, \ldots, X_n)$;
while $\neg\mathrm{halt}(A)$ **do** $A := \mathrm{next}(A)$;
$X_{n+1} := \mathrm{res}^{(n)}(A)$

Of course the identifier A must be different from X_0, …, X_{n+1}. Note that this program still yields the right result if u is not the code number of a $\mathcal{WL}$-program. In this case halt(A) already holds after the initial assignment $A := \mathrm{init}^{(n)}(\mathrm{pl}(X_0), X_1, \ldots, X_n)$, and the component of A which represents X_{n+1} is left with the value 0. □

Exercise 3.2: [2] Show that for every n there exists a fixed number M such that every $f \in \mathcal{P}^{(n)}$ is computed by some URM with no more than M registers. (You are not required to determine M.)

Theorem 3.2 is strengthened by the following result which is called the *normal form theorem for partial recursive functions:*

Theorem 3.3: There exist a fixed function $d \in \mathcal{PR}$ and for every n a fixed predicate $T^{(n)} \in \mathcal{PR}_*$ such that for all u, $\vec{x}_n$

$$\phi_u^{(n)}(\vec{x}_n) = d\big((\mu z)[T^{(n)}(z, u, \vec{x}_n)]\big);$$

moreover, for every n and $\vec{x}_n$ the condition $T^{(n)}(z, u, \vec{x}_n)$ holds for at most one z.

Exercise 3.3: [0] Explain why theorem 3.2 is an immediate consequence of theorem 3.3.

The predicates $T^{(n)}$ to be constructed are in fact of considerable independent importance and will be used in the proofs of several theorems. For $n = 1$ it is called the *Kleene predicate*. Here again we omit the superscript (n), if it equals 1.

Proof of theorem 3.3: The predicate $T^{(n)}(\langle s, y\rangle, u, \vec{x}_n)$ has the following informal meaning: it asserts that the computation of $\phi_u^{(n)}(\vec{x}_n)$, that is, the computation of the $\mathcal{WL}$-program with code number u, terminates after exactly s steps and yields the result y. Let $\mathrm{conf}^{(n)}(u, \vec{x}_n, s)$ be the result of s steps of this computation:

$$\mathrm{conf}^{(n)}(u, \vec{x}_n, s) = \mathrm{next}^s\big(\mathrm{init}^{(n)}(\mathrm{pl}(u), \vec{x}_n)\big).$$

The function $\mathrm{conf}^{(n)}$ is primitive recursive (example 2.4 (vi)). Now we get

$$\begin{aligned} T^{(n)}(v, u, \vec{x}_n) \equiv{}& L(v) = 2 \mathbin{\&} \mathrm{halt}\big(\mathrm{conf}^{(n)}(u, \vec{x}_n, v.0)\big) \\ &\mathbin{\&}(\forall z < v.0)\big(\neg\mathrm{halt}(\mathrm{conf}^{(n)}(u, \vec{x}_n, z))\big) \mathbin{\&} \mathrm{res}\big(\mathrm{conf}^{(n)}(u, \vec{x}_n, v.0)\big) = v.1. \end{aligned}$$

If $\phi_u^{(n)}(\vec{x}_n)\uparrow$, then $\mathrm{conf}^{(n)}(u, \vec{x}_n, s)$ is not a halting configuration for any s, hence $(\mu z)[T^{(n)}(z, u, \vec{x}_n)]\uparrow$. If $u = \mathrm{cd}_{\mathcal{P}}(P)$ for a $\mathcal{WL}$-program P and the execution of P for input $\vec{x}_n$ yields output y after s steps, then $(\mu z)[T^{(n)}(z, u, \vec{x}_n)] = \langle s, y\rangle$. Finally, if u is not the code of a $\mathcal{WL}$-program, then $(\mu z)[T^{(n)}(z, u, \vec{x}_n)] = \langle 0, 0\rangle$. In each case $\phi_u^{(n)}(\vec{x}_n) = \big((\mu z)[T^{(n)}(z, u, \vec{x}_n)]\big).1$, hence the assertion holds with $d = \lambda v.(v.1)$. □

We now come to the second fundamental result of this section. In essence, it asserts the following: If a partial recursive function g is obtained from f by substituting constants for some of the arguments, so that the remaining arguments of f are the arguments of g, then an index of g can be computed effectively from an index of f and from the constants substituted.

Theorem 3.4 (S-m-n-theorem): For all m, n there exists a one-one primitive recursive function (P) s_{mn} such that for all u, $\vec{x}_m$, $\vec{y}_n$,

$$\phi^{(n)}_{s_{mn}(u, \vec{x}_m)}(\vec{y}_n) = \phi_u^{(m+n)}(\vec{x}_m, \vec{y}_n).$$

Proof: We construct a one-one primitive recursive function $f\colon \omega^{**} \times \omega^m \to \omega^{**}$, considered as a function of $m+1$ arguments of types ω^{**}, ω, …, ω, such that if P computes $\phi_u^{(m+n)}$ with respect to X_1, …, X_{m+n+1}, then $f(\hat{P}, \vec{x}_m) = \hat{P}'$, where P' computes $\lambda\vec{y}_n.\phi_u^{(m+n)}(\vec{x}_m, \vec{y}_n)$ with respect to X_1, …, X_n, X_{n+1}. Then f_ω serves as s_{mn}; since $\mathrm{cd}_{\omega^{**}}$ is onto, the property of being one-one carries over from f to s_{mn}.

In order to obtain P' from P one has to permute the names of the variables, renaming X_{m+1}, …, X_{m+n+1} to X_1, …, X_{n+1}, and X_1, …, X_m to X_{n+2}, …, X_{m+n+1}.

Also, one has to put additional statements at the beginning of P', which initialize X_{n+2}, $\ldots$, X_{m+n+1} to x_1, $\ldots$, x_m, followed by the statement $X_0 := X_0 \dot{-} 1$. The latter does not change the value of X_0 (which was already 0), but it serves to mark the beginning of the portion of P' which is obtained from P by renaming variables; without them f would not be one-one.

Let

$$g(x) = \begin{cases} x + n + 1 & \text{if } 1 \le x \le m, \\ x \dot{-} m & \text{if } m + 1 \le x \le m + n + 1, \\ x & \text{otherwise,} \end{cases}$$

and let $h\colon \omega^* \to \omega^*$ be defined by

$$h(x) = \begin{cases} \mathrm{st}\big(x, 1, g(x[1])\big) & \text{if } L(x) \ge 2, \\ x & \text{if } L(x) \le 1. \end{cases}$$

Thus if I is a $\mathcal{WL}$-statement, then $h(\hat{I}) = \hat{I}'$, where I' is obtained from I be renaming variables as described above. Now the following $\mathcal{AL}$-program computes f with respect to U, V_1, $\ldots$, V_m and $U1$:

$U, U1\colon \omega^{**}$;
$S := V_1 + \cdots + V_m$;
for $I := 1$ **to** V_1 **do** $U1[I \dot{-} 1] := (0, n+2)$;
for $I := V_1 + 1$ **to** $V_1 + V_2$ **do** $U1[I \dot{-} 1] := (0, n+3)$;
$\vdots$
for $I := V_1 + \cdots + V_{m-1} + 1$ **to** S **do** $U1[I \dot{-} 1] := (0, m+n+1)$;
$U1[S] := (1, 0)$;
for $I := 1$ **to** $L(U)$ **do** $U1[I + S] := h(U[I \dot{-} 1])$

This construction also works if U is *not* of the form $\hat{P}$ for any $\mathcal{WL}$-program P. In this case $f(U, \vec{x}_m)$ is not of this form either; the property of U of being or not being the list representation of some $\mathcal{WL}$-program is not altered by inserting codes of assignment statements and changing $U[i, 1]$ for some i. Hence if u is the code of U and u' that of $f(U, \vec{x}_m)$, then $\phi_u^{(m+n)}(\vec{x}_m, \vec{y}_n)$ and $\phi_{u'}^n(\vec{y}_n)$ are both 0 for all $\vec{x}_m$, $\vec{y}_n$.

It only remains to show that f is one-one, that is, U, x_1, $\ldots$, x_m are uniquely determined from $U' = f(U, \vec{x}_m)$. U' begins with x_1 copies of $(0, n+2)$ followed by x_2 copies of $(0, n+3)$ etc., followed by x_m copies of $(0, m+n+1)$, followed by $(1, 0)$; this uniquely determines x_1, $\ldots$, x_m. Also, if s is minimal such that $U'[s, 1] = 0$, then $L(U) = L(U') - (s+1)$ and $U[i] = h^{-1}(U'[i+s+1])$ for $i < L(U') - (s+1)$. Note that the function h is bijective. However, if $f(U, \vec{x}_m)$ had been defined otherwise, without inserting $(1, 0)$ at component s, then there could be more than x_m copies of $(0, m+n+1)$ starting at component $x_1 + \cdots + x_{m-1}$ of U' (namely if $U[0] = (0, m)$, so that $h(u[0]) = (0, m+n+1)$). Hence the reconstruction of U and x_m would not work in this case. $\square$

Exercise 3.4: [2] Suppose we had defined $h(x) = \mathrm{st}\big(x, 1, g(x[1])\big)$ for all x, even if $L(x) \le 1$. Would the proof of theorem 3.4 still work?

Applications of the S-m-n-theorem almost always take the following form:

Corollary 3.5: If $g(\vec{x}_m, \vec{y}_n)$ is partial recursive, then there exists a one-one, primitive recursive function f such that for all $\vec{x}_m$, $\vec{y}_n$

$$\phi^{(n)}_{f(\vec{x}_m)}(\vec{y}_n) = g(\vec{x}_m, \vec{y}_n).$$

Proof: Let u be an index of g, that is, $g = \phi^{(m+n)}_u$. Then $f(\vec{x}_m) = s_{mn}(u, \vec{x}_m)$ satisfies the required conditions. □

In general corollary 3.5 allows to show results of the following kind: If g is obtained by certain operations from $g_1, g_2, \ldots$, then an index of g can be computed effectively from indices of $g_1, g_2, \ldots$. This is in fact the problem of program transformation, except that one works not on the programs themselves, but on their code numbers. Here is an example:

Example 3.1: There exists a primitive recursive function f such that for all x_1, x_2

$$\phi_{f(x_1,x_2)} = \lambda y.\phi_{x_1}\big(\phi_{x_2}(y)\big) = \phi_{x_1}\phi_{x_2}.$$

(Recall that the upper index in $\phi^{(n)}_u$ is omitted, if it is 1, and likewise for $\psi^{(n)}$.)

Proof: Apply corollary 3.5 to $g = \lambda x_1 x_2 y.\phi_{x_1}\big(\phi_{x_2}(y)\big) = \lambda x_1 x_2 y.\psi\big(x_1, \psi(x_2, y)\big)$. □

Exercise 3.5: [1] Show that there exist primitive recursive functions f, g, h such that for all x, y, z

(i) $\phi_{f(x,y)}(z) = \phi_x(z) + \phi_y(z)$;
(ii) $\phi_{g(x,y)}(z) = \phi^y_x(z)$;
(iii) $\phi_{h(x,y)}(z) = \phi_{\phi_z(x)}(y)$.

It is easy to see that every partial recursive function has not just one, but infinitely many indices. The next theorem shows that an infinite sequence of indices of $\phi^{(n)}_u$ can be computed effectively from u; in fact, a little more is shown.

Theorem 3.6: There exists a one-one primitive recursive function (P) ind such that for all n, u, s

$$\phi^{(n)}_{\mathrm{ind}(u,s)} = \phi^{(n)}_u.$$

Proof: In analogy to the proof of theorem 3.4 we construct a one-one primitive recursive function $f\colon \omega^{**} \times \omega \to \omega^{**}$ such that for every $\mathcal{WL}$-program P and every s, $f(\hat{P}, s) = \hat{P}'$, where P' computes the same function as P with respect to $X_1, \ldots, X_n, X_{n+1}$ for every n. P' is obtained from P by attaching the instructions $X_s := X_s + 1$; $X_s := X_s \dot{-} 1$. The following $\mathcal{AL}$-program computes f with respect to U, S and $U1$:

U, $U1\colon \omega^{**}$;
$U1 := U$; $L := L(U)$; $U1[L] := (0, S)$; $U1[L+1] := (1, S)$

Then $\mathrm{ind} = f_\omega$. As in the proof of theorem 3.4 the condition $\phi^{(n)}_{\mathrm{ind}(u,s)} = \phi^{(n)}_u$ still holds if u is not the code number of a $\mathcal{WL}$-program; in this case $\phi^{(n)}_{\mathrm{ind}(u,s)} = \phi^{(n)}_u = \lambda\vec{x}_n.0$. □

The proofs of theorems in the rest of these lecture notes (except for chapter 4) differ considerably from most proofs that occured so far. They are based on theorems 3.2–3.4, and on the fact that certain specific functions are (primitive/partial) recursive, but they do not depend on one specific definition of $\mathcal{PR}$, $\mathcal{R}$ and $\mathcal{P}$, nor on our specific enumerations of $\mathcal{P}^{(n)}$, $n \in \omega$. This is expressed by saying that these proofs are *formalism-independent*. For the further development of the theory any family of enumerations of the classes $\mathcal{P}^{(n)}$ $(n \in \omega)$ could serve as a basis, provided that it satisfies theorems 3.2–3.4. It is in this sense that theorems 3.2–3.4 are the fundamental ones; they may be regarded as a system of axioms for enumerations of $\mathcal{P}^{(n)}$.

The proof of corollary 3.5 and example 3.1 are already simple examples of formalism-independent proofs. The proof of theorem 3.6 had to be formalism-dependent only because we wished to show that the function ind satisfying $\phi^{(n)}_{\mathrm{ind}(u,s)} = \phi^{(n)}_u$ is independent of n.

Exercise 3.6: [2] Give a formalism-independent proof of the fact that for every n there exists a one-one primitive recursive function $\mathrm{ind}^{(n)}$ such that for all u, s

$$\phi^{(n)}_{\mathrm{ind}^{(n)}(u,s)} = \phi^{(n)}_u.$$

Enumerations of $\mathcal{P}^{(n)}$ will be considered again in section 6.3.

3.2. $\mathcal{PR} \neq \mathcal{R}$.

As promised at the beginning of this chapter, we now use the method of diagonalization for the construction of recursive functions which are not primitive recursive. Such a function can behave in various ways, but two extreme possibilities are typically considered: f may grow very fast, faster than every primitive recursive function, or f may only take the values 0 and 1.

Definition 3.3: Let f and g be total functions from ω to ω. f *dominates* g, iff there exists a number N such that $f(x) > g(x)$ for all $x \geq N$.

Exercise 3.7: [2] In analysis the phrase 'f grows faster than g' usually means that $\lim_{x\to\infty} g(x)/f(x) = 0$; this condition, denoted by $g = o(f)$, is stronger than that of definition 3.3. Show, however, that $f\colon \omega \to \omega$ dominates every $g \in \mathcal{PR}^{(1)}$ if and only if $g = o(f)$ for every $g \in \mathcal{PR}^{(1)}$.

Theorem 3.7: (i) There exists a (1-place) recursive function which dominates every 1-place primitive recursive function.

(ii) There exists a function $f \in \mathcal{R}^{(1)}$ with range $\{0, 1\}$ which is not primitive recursive.

(iii) $\mathcal{PR} \subsetneq \mathcal{R}$.

(iv) $\mathcal{PR}_* \subsetneq \mathcal{R}_*$.

Proof: (i) We introduce an enumeration ϕ'_0, ϕ'_1, ... for $\mathcal{PR}^{(1)}$ and a corresponding universal function ψ'. Let $\phi'_u = \phi_u$ if u is the code number of some special $\mathcal{WL}$-program, and $\phi'_u = \lambda x.0$ otherwise. Also, let $\psi' = \lambda ux.\phi'_u(x)$, so that

$$\psi'(u,x) = \begin{cases} \psi(u,x) & \text{if } \mathrm{swl}(\mathrm{pl}(u)), \\ 0 & \text{otherwise.} \end{cases}$$

This representation shows that ψ' is partial recursive; in fact, ψ' is recursive, since ψ' is clearly total.

Now let $f(x) = \sum_{i \le x} \phi'_i(x) + 1 = \sum_{i \le x} \psi'(i,x) + 1$. Fix i. We have $f(x) > \phi'_i(x)$ for all $x \ge i$, hence f dominates ϕ'_i. Since this holds for every $i \in \omega$, f dominates every 1-place primitive recursive function. This argument also shows that ψ' is not primitive recursive, otherwise f would be primitive recursive.

(ii) Let $f(x) = 1 \dot{-} \phi'_x(x) = 1 \dot{-} \psi'(x,x)$. Then $f(x) \ne \phi'_x(x)$, that is, $f \ne \phi'_x$, for all x.

(iii) This is an immediate consequence of (i) or (ii).

(iv) The function f from (ii) is the characteristic function of a predicate P which is recursive, but not primitive recursive. □

At this point it is time for a more detailed discussion of diagonalization. In the purest form this method occurred in the proof of (ii). The function f had to satisfy the requirements $f \ne \phi'_x$ $(x \in \omega)$, and the idea was simply to define f such that f differs from ϕ'_x for argument x, that is, $f(x) \ne \phi'_x(x)$. This and the additional requirement $\operatorname{ran} f = \{0,1\}$ are satisfied by defining $f(x) = 1 \dot{-} \phi'_x(x)$. In addition, f happens to be recursive, due to the fact that ψ' is recursive.

In general this method allows to construct a function $f\colon \omega \to \omega$ which differs from each of countably many functions $f_i\colon \omega \to \omega$. The term *diagonalization* stems from the fact that f depends on the values $f_x(x)$, which appear in the diagonal of the array $\big(f_x(y)\big)$ $(x, y \in \omega)$:

$$\begin{array}{llll}
f_0(0) \ne f(0) & f_0(1) & f_0(2) & \dots \\
f_1(0) & f_1(1) \ne f(1) & f_1(2) & \dots \\
f_2(0) & f_2(1) & f_2(2) \ne f(2) & \dots \\
\vdots & \vdots & \vdots & \ddots
\end{array}$$

In fact, the given objects f_i and the object f do not have to be functions; they may also be other types of mathematical objects A which are characterized by countably many pieces of information $A.x$ $(x \in \omega)$. For instance, $A.x$ may be the x-th digit of A if $A \in \mathbb{R}$, or the information whether or not $x \in A$ if $A \subseteq \omega$. Corresponding diagonalization proofs yield two well-known theorems by Cantor which state that $\mathbb{R}$ and 2^ω are uncountable; the latter already occurred in section 3.1.

What has just been described is the method of diagonalization as it is typically characterized in textbooks, see, for instance, Odifreddi [17], p. 145, or Tourlakis [25], p. 23 and p. 283. However, there are numerous variations, some trivial, some substantial. In what follows we assume that a mathematical object A has to be constructed which is either a countable set or a function with countable domain. A has to satisfy a countable set of conditions $\mathcal{C}_i$ $(i \in \omega)$ which are not necessarily of the form $A \ne A_i$ for given objects A_i. To define A at a place a means: Define $A(a)$ if A is a function, or decide whether or not $a \in A$ if A is a set.

In the simplest case $\mathcal{C}_i$ can be guaranteed by defining A appropriately at one particular place a_i; in this case we say that $\mathcal{C}_i$ is satisfied at a_i. So far we had $a_i = i$, but sometimes a different choice may be necessary, depending on $\mathcal{C}_i$. Normally all a_i have to be different, for if $a = a_i = a_j$ with $i \ne j$, then $\mathcal{C}_i$ and $\mathcal{C}_j$ may put conflicting requirements on the definition of A at a. In other circumstances there may be finite sets E_i such that $\mathcal{C}_i$ is guaranteed by defining A appropriately at all $a \in E_i$. Then the proof is still straightforward if the sets E_i can be chosen pairwise disjoint.

The proof of theorem 3.7 (i) uses a slightly more complicated variant. Let $\mathcal{C}_i$ be the condition that f dominate ϕ'_i. Then $\mathcal{C}_i$ cannot be satisfied at just one place; instead it influences the definition of $f(x)$ for almost all x. More precisely, the definition of $f(x)$ takes the condition $\mathcal{C}_i$ into account for all $i \leq x$. This works, since only finitely many conditions have to be taken into account for every x, while on the other hand every condition $\mathcal{C}_i$ is ignored only a finite number of times.

The next level of complexity occurs when it is not known in advance where and how $\mathcal{C}_i$ can be satisfied for each i. A is constructed in steps, and one has to be prepared to satisfy $\mathcal{C}_i$ as soon as an occasion arises. This is how one has to proceed for exercise 3.10 below, for instance.

Finally, it may happen that certain conditions $\mathcal{C}_i$ are satisfied, then violated, then satisfied again etc. This happens in proofs using the *priority method*, which will be discussed in (DCC).

Exercise 3.8: Let $\mathcal{F}$ be a countable set of total functions from ω to ω, like $\mathcal{R}^{(1)}$.

(i) [0] Show that there exists a function $g: \omega \to \omega$ which dominates every $f \in \mathcal{F}$.

(ii) [0] Show that there exists a function $h: \omega \to \omega$ such that $h \neq f$ for all $f \in \mathcal{F}$ and $\operatorname{ran} h = \{0, 1\}$.

(iii) [0] Suppose that there exists an enumeration $(f_0, f_1, \ldots)$ of $\mathcal{F}$ such that the corresponding universal function $\lambda ux.f_u(x)$ is (primitive) recursive. Show that the functions g and h from (i), (ii) can be made (primitive) recursive.

(iv) [3] Let $\mathcal{F}_0$ be a finite set of recursive functions, and let $\mathcal{F}$ be the set of all functions which are computed by some $\mathcal{AL}(\mathcal{F}_0)$-program without **while**- and **goto**-statements and recursive procedures. Define an enumeration $(f_0, f_1, \ldots)$ of $\mathcal{F}$ such that $\lambda ux.f_u(x)$ is recursive, so that (iii) applies.

Exercise 3.9: [2] Define properties P_1, P_2, P_3 for functions $f: \omega^n \to \omega$ as follows:

(a) f satisfies P_1, iff f is primitive recursive;

(b) f satisfies P_2, iff f is recursive, but not primitive recursive;

(c) f satisfies P_3, iff f is total, but not recursive.

Let i, j, k be (not necessarily different) elements of $\{1, 2, 3\}$ such that $i \leq k$, $j \leq k$. Show that there exists a function $f(x, y)$ with the following properties:

(i) f satisfies P_k;

(ii) $\lambda y.f(x, y)$ satisfies P_i for all x;

(iii) $\lambda x.f(x, y)$ satisfies P_j for all y.

(This problem can be solved very quickly, once the right idea has been found.)

Exercise 3.10: [6] Show that there exists a recursive set $A \subset \omega$ such that neither A nor $\overline{A}$ contains an infinite primitive recursive subset. (This is definitively one of those exercises which are easier to do later, when some more techniques for constructing sets with certain properties have been presented. See, for example, theorem 6.16.)

The proof of theorem 3.7 results naturally from the development of the theory of recursive functions. Another example of a recursive function which is not primitive recursive is the *Ackermann function.* To introduce it, first define a sequence of functions (P) A_0, A_1, ... such that for each i A_{i+1} is obtained by iterating A_i and grows faster than A_i:

$$A_0(x) = x + 2,$$
$$A_{i+1}(x) = A_i^x(1).$$

In particular

$$A_1 = \lambda x.2x + 1,$$
$$A_2 = \lambda x.2^{x+1} - 1.$$

There is no closed-form representation for A_3, but its rate of growth can be estimated by noting that

$$A_3' = \lambda x.\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}x$$

is the function obtained by iterating $\lambda x.2^x$ instead of A_2; thus we can conclude that $A_3(x)$ is somewhat larger than $A_3'(x)$.

By induction on i each of the functions A_i is still primitive recursive. The function A_3 already grows at a tremendous rate, and each of the functions A_4, A_5 etc. grows very much faster than the previous one. Thus the assertion that there exist recursive functions which dominate every $f \in \mathcal{PR}^{(1)}$ is a dramatic one indeed.

The Ackermann function itself is obtained from the functions A_i by making i an argument:

$$\text{(P) } A = \lambda ix.A_i(x).$$

Exercise 3.11: (i) [8] Show that $\lambda x.A(x, x)$ dominates every 1-place primitive recursive function.

(ii) [0] Conclude that A is not primitive recursive.

The assertion of exercise 3.11 (i) will be shown in (DCC); this proof would not fit well at the present place. We note in passing that the definition of the functions A_i and $A = \lambda ix.A_i(x)$ is of type (V). It varies slightly from textbook to textbook, but in each case A_{i+1} is obtained by iterating A_i.

3.3. Basic properties of recursively enumerable sets.

Up to this point the only significant result about r.e. sets was theorem 1.3 (ii). Other proofs often require a formalization of certain intuitive ideas like that of carrying out several computations simultaneously. This requires the tools from section 3.1; in particular, the Kleene predicate plays a crucial rôle, as will be seen. We start by presenting some alternative conditions (other than that of definition 1.6) that characterize r.e. sets.

Theorem 3.8: Let $A \subseteq \omega^n$. The following assertions are equivalent:

(i) $A = \emptyset$ or A is the range of some recursive function $f: \omega \to \omega^n$.

(ii) A is the range of some partial recursive function $f: \omega \to \omega^n$.

(iii) A is r.e., that is, A is the domain of some partial recursive function.

(iv) There exists a recursive predicate $R(y, \vec{x}_n)$ such that

$$A = \{\, \vec{x}_n \mid (\exists y)\big(R(y, \vec{x}_n)\big) \,\}.$$

Condition (i) yields a procedure for *generating* A (in the intuitive sense). Indeed, if A is the range of a recursive function f, then the procedure consists simply in outputting $f(0)$, $f(1)$, ... in order. Conversely, assume that there is a procedure generating A, giving x_0, x_1, ... in order as outputs. Unless $A = \emptyset$, we may assume that x_i exists for all i, since some x_j may be repeated, if necessary. Now the function $f = \lambda i.x_i$ is surely effectively computable, and $A = \operatorname{ran} f$. This argument will be formalized in theorem 3.14; the latter result provides yet another criterion for A being r.e., which comes even closer to the intuition of A being generated by a procedure.

Condition (ii) is just a variant of (i). As noted in section 1.3, condition (iii) means that A is accepted by some procedure, whereas (iv) is yet another very important characterization of r.e. sets, which will be discussed further below.

Proof of theorem 3.8: (i) $\Longrightarrow$ (ii): If A is the range of a recursive function $f\colon \omega \to \omega^n$, then we are done. If $A = \emptyset$, then A is the range of the nowhere defined function $f\colon \omega \to \omega^n$; f is partial recursive by exercises 2.13 and 1.5 (i).

(ii) $\Longrightarrow$ (iii): First we reduce the general case to the case $n = 1$. Suppose that $n > 1$ and $A = \operatorname{ran} f$. Then $\iota_n(A) = \operatorname{ran}(\iota_n f)$, hence $\iota_n(A)$ is r.e., assuming that the assertion holds for $n = 1$. By theorem 2.9 A is also r.e.

Now assume $n = 1$. Let A be the range of a partial recursive function $f\colon \omega \to \omega$, and let u be an index of f, so that $f = \phi_u = \lambda y.\big((\mu z)[T(z,u,y)]\big).1$ (theorem 3.3). Then for all x we have

$$x \in A \iff (\exists z, y)\big(T(z,u,y)\ \&\ z.1 = x\big) \iff (\exists v)\big(T(v.0,u,v.1)\ \&\ v.0.1 = x\big).$$

Hence A is the domain of $\lambda x.(\mu v)[T(v.0,u,v.1)\ \&\ v.0.1 = x]$.

(iii) $\Longrightarrow$ (iv): If $A = \operatorname{dom}(\phi_u^{(n)})$, then

$$\vec{x}_n \in A \iff (\exists y)\big(T^{(n)}(y,u,\vec{x}_n)\big).$$

(iv) $\Longrightarrow$ (i): If $A = \emptyset$, then we are done, so assume $A \neq \emptyset$. Once again the general case is reduced to the case $n = 1$. Let $A = \{\, \vec{x}_n \mid (\exists y)\big(R(y,\vec{x}_n)\big) \,\}$, where $R \in \mathcal{R}_*$. Then $\iota_n(A) = \{\, x \mid (\exists y)\big(R(y,\kappa_1^n(x),\ldots,\kappa_n^n(x))\big) \,\}$, hence, if the assertion holds for $n = 1$, $\iota_n(A) = \operatorname{ran} f$ for some recursive function f. Hence $A = \operatorname{ran} g$, where $g(x) = \big(\kappa_1^n(f(x)),\ldots,\kappa_n^n(f(x))\big)$.

Now let $n = 1$, and let $A = \{\, z \mid (\exists y)\big(R(y,z)\big) \,\}$, with $R \in \mathcal{R}_*$. We assumed $A \neq \emptyset$, so let a be a fixed element of A. A function $f \in \mathcal{R}^{(1)}$ with range A is defined as follows:

$$f(x) = \begin{cases} \kappa_2^2(x) & \text{if } R\big(\kappa_1^2(x),\kappa_2^2(x)\big), \\ a & \text{if } \neg R\big(\kappa_1^2(x),\kappa_2^2(x)\big). \end{cases}$$

Indeed, if $z \in A$, then $R(y,z)$ holds for some y, whereupon $f\big(\iota_2(y,z)\big) = z$. Conversely, if $z \in \operatorname{ran} f$, say, $z = f(x)$, then either $z = a$, or $x = \iota_2(y,z)$, where $R(y,z)$ holds; in both cases $z \in A$. $\square$

Exercise 3.12: [0] Show that the following conditions are also equivalent to $A \subseteq \omega^n$ being r.e.:

(v) Condition (i) or (ii) with the modification that the function f is from ω^m to ω^n, where $m \geq 1$ is given.

(vi) The function f defined by

$$f(\vec{x}_n) = \begin{cases} 0 & \text{if } \vec{x}_n \in A, \\ \uparrow & \text{if } \vec{x}_n \notin A \end{cases}$$

is partial recursive.

(vii) Condition (iv) from theorem 3.9 with the stronger requirement that R be primitive recursive.

(viii) Condition (i) from theorem 3.9 with the stronger requirement that f be primitive recursive in the case $A \neq \emptyset$.

Example 3.2: Define

$$\text{(P)}\ K_1 = \{\, (x, y, z) \mid \phi_x(y) = z \,\}.$$

The set K_1 is r.e., since

$$K_1 = \left\{\, (x, y, z) \mid (\exists u)\big([T(u, x, y)\ \&\ u.1 = z]\big) \,\right\}.$$

Definition 3.4: $W_u^{(n)}$ is defined to be the set $\mathrm{dom}(\phi_u^{(n)})$; as usual, the superscript (n) is left out if it is 1. If $A = W_u^{(n)}$, then u is called an *(r.e.) index* of A.

Thus $W_0^{(n)}$, $W_1^{(n)}$, ... is an enumeration of $\mathcal{E}^{(n)}$. It is clear that every r.e. set, like every partial recursive function, has infinitely many indices, and that such indices are obtained by the function ind of theorem 3.6:

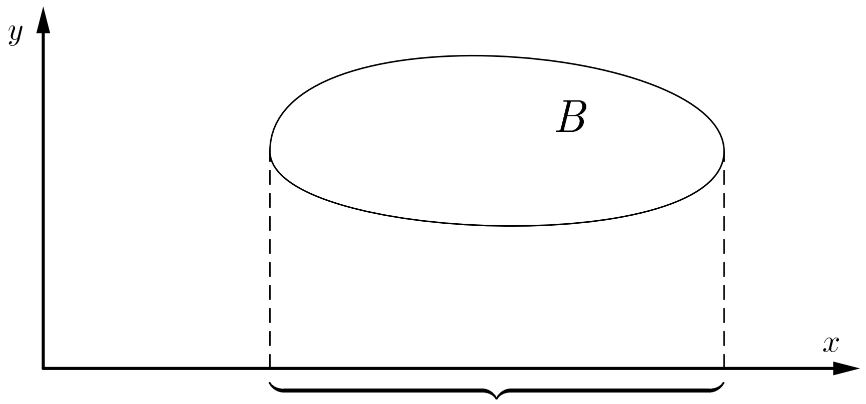
$$(\forall x, s)\big(W_{\mathrm{ind}(x,s)}^{(n)} = W_x^{(n)}\big).$$

There is a general name for the operation by which A is obtained from R in part (iv) of theorem 3.8:

Definition 3.5: $A \subseteq \omega^n$ is obtained from $B \subseteq \omega^{n+1}$ by *projection*, iff

$$A = \left\{\, \vec{x}_n \mid (\exists y)\big((y, \vec{x}_n) \in B\big) \,\right\}.$$

Geometrically, the situation may be pictured as follows:



$$A = \left\{\, x \mid (\exists y)\big((y, x) \in B\big) \,\right\}$$

Thus an r.e. set is obtained by applying projection to a recursive set. The next theorem shows that applying projection to an r.e. set once again yields an r.e. set:

Theorem 3.9: Let $B \in \mathcal{E}^{(n+1)}$, $A = \{\, \vec{x}_n \mid (\exists y)\big((y, \vec{x}_n) \in B\big) \,\}$. Then A is also r.e.

Proof: Since B is r.e., we have $B = \{\, (y, \vec{x}_n) \mid (\exists z)\big((z, y, \vec{x}_n) \in C\big) \,\}$ for some recursive set C. Then

$$A = \{\, \vec{x}_n \mid (\exists y)(\exists z)\big((z, y, \vec{x}_n) \in C\big) \,\} = \{\, \vec{x}_n \mid (\exists u)\big((u.0, u.1, \vec{x}_n) \in C\big) \,\}.$$

Here the predicate $\lambda u\vec{x}_n.(u.0, u.1, \vec{x}_n \in C)$ is recursive. Hence A is obtained by projection from a recursive set, and A is r.e. □

Like (primitive/partial) recursive functions and (primitive) recursive sets, the class of r.e. sets is closed under various operations. Such closure properties will be investigated systematically in chapter 5. At the moment we prove just one result, and a few more are provided in the next exercise.

Theorem 3.10: If $A \subseteq \omega^n$ is r.e., and $f_1(\vec{x}_m)$, $\ldots$, $f_n(\vec{x}_m)$ are partial recursive functions, then the set

$$B = \{\, \vec{x}_m \mid f_1(\vec{x}_m)\downarrow \,\&\, \ldots \,\&\, f_n(\vec{x}_m)\downarrow \,\&\, \big(f_1(\vec{x}_m), \ldots, f_n(\vec{x}_m)\big) \in A \,\}$$

is also r.e.

Proof: Apply the original definition of r.e. sets from definition 1.7. If A is the domain of g, then B is the domain of $\lambda\vec{x}_m.g\big(f_1(\vec{x}_m), \ldots, f_n(\vec{x}_m)\big)$. □

From now on results on r.e. sets $A \subseteq \omega^n$ will be formulated only for $n = 1$; it is a trivial matter to extend them to $n > 1$, using invariance under coding.

Exercise 3.13: [2] Show that if A and $B \subseteq \omega$ are r.e., then so are the following sets:
(i) $A \cup B$;
(ii) $A \cap B$;
(iii) (P) $A \times B = \{\, (x, y) \mid x \in A \,\&\, y \in B \,\}$;
(iv) $\{\, x \mid \big(\forall z < L(x)\big)(x.z \in A) \,\}$.
These results will be considerably extended and strengthened in theorem 5.10 and exercise 5.5.

The relation between recursive and r.e. sets is given by the following

Theorem 3.11: A set $A \subseteq \omega$ is recursive if and only if A and $\overline{A}$ are both r.e.

Proof: $\Longrightarrow$: Assume that A is recursive. Then $\overline{A}$ is also recursive (example 2.3 (iii)), hence A and $\overline{A}$ are r.e. by theorem 1.3 (ii).

$\Longleftarrow$: Assume that A and $\overline{A}$ are both r.e. Informally, it can be decided for given x whether or not $x \in A$ by simultaneously enumerating A and $\overline{A}$ until x appears in one of these enumerations. A formal argument runs as follows: If $A = \emptyset$ or $A = \omega$, then we are done. Otherwise let f_0, f_1 be recursive functions such that $\mathrm{ran}(f_0) = A$, $\mathrm{ran}(f_1) = \overline{A}$. Define

$$f(x) = \begin{cases} f_0(x/2) & \text{if } x \text{ is even,} \\ f_1(\lfloor x/2 \rfloor) & \text{if } x \text{ is odd,} \end{cases}$$

$$g(x) = (\mu z)[f(z) = x].$$

g is total, since every x belongs either to A or to $\overline{A}$. Now we have

$$x \in A \iff g(x) \text{ is even.}$$

$\square$

Exercise 3.14: [2] Assume that $A_1, \ldots, A_n \subseteq \omega$ are pairwise disjoint r.e. sets, and that $\bigcup_i A_i$ is recursive. Show that all A_i are recursive. Why is this a generalization of theorem 3.11?

It will be shown at the end of this section that there exist r.e. sets which are not recursive; this is left open by theorem 3.11. But first we prove two more theorems, thus supplementing results from chapter 2. The next one is the generalization of theorem 2.5.

Theorem 3.12: Let $A_1, \ldots, A_k \subseteq \omega^n$ be disjoint r.e. sets, $f_1, \ldots, f_k$ partial recursive functions. Then the function f defined by

$$f(\vec{x}_n) = \begin{cases} f_1(\vec{x}_n) & \text{if } \vec{x}_n \in A_1, \\ \vdots & \\ f_k(\vec{x}_n) & \text{if } \vec{x}_n \in A_k, \\ \uparrow & \text{otherwise} \end{cases}$$

is partial recursive.

Proof: For $1 \leq i \leq k$ let g_i be a partial recursive function with domain A_i,

$$f_i'(\vec{x}_n) = f_i(\vec{x}_n) + 0 \cdot g_i(\vec{x}_n) = \begin{cases} f_i(\vec{x}_n) & \text{for } \vec{x}_n \in A_i, \\ \uparrow & \text{for } \vec{x}_n \notin A_i, \end{cases}$$

let u_i be an index of f_i', and finally, define

$$h(\vec{x}_n) = (\mu z)[T^{(n)}(z, u_1, \vec{x}_n) \vee \cdots \vee T^{(n)}(z, u_k, \vec{x}_n)].$$

Then

$$f(\vec{x}_n) = \big(h(\vec{x}_n)\big).1.$$

Indeed, $h(\vec{x}_n)$ is undefined for $\vec{x}_n \notin \bigcup_i A_i$. But if $\vec{x}_n \in A_i$, then $T^{(n)}(z, u_j, \vec{x}_n)$ cannot hold for $j \neq i$, hence

$$\big(h(\vec{x}_n)\big).1 = \big((\mu z)[T^{(n)}(z, u_i, \vec{x}_n)]\big).1 = f_i'(\vec{x}_n) = f_i(\vec{x}_n).$$

$\square$

In example 2.3 (iv) we raised the question how a function f is related to its graph with respect to computability properties. The following theorem, which summarizes some new facts and some facts already known, provides a complete answer.

Theorem 3.13: (i) Let $f\colon \omega \to \omega$ be total. If f is primitive recursive, then so is $\mathrm{gr}(f)$, but the converse does not always hold.

(ii) Assume again that f is total. Then

$$f \text{ is recursive} \iff \mathrm{gr}(f) \text{ is recursive} \iff \mathrm{gr}(f) \text{ is r.e.}$$

(iii) Let f be arbitrary. Then

$$f \text{ is partial recursive} \iff \mathrm{gr}(f) \text{ is r.e.}$$

Proof: (i) We know already (example 2.2) that for $f \in \mathcal{PR}^{(1)}$ the predicate $\lambda xy.f(x) = y$ is primitive recursive; this is equivalent to saying that $\mathrm{gr}(f)$ is primitive recursive. We now show that the converse does not necessarily hold. Let ϕ_u be a 1-place recursive function which is not primitive recursive (theorem 3.7). We have $\phi_u(x) = \big((\mu z)[T(z,u,x)]\big).1$. Let $f(x) = (\mu z)[T(z,u,x)]$. Then f is recursive, but not primitive recursive, for if f were non-total or primitive recursive, then the same would follow for ϕ_u. On the other hand, $\mathrm{gr}(f) = \lambda xz.T(z,u,x)$ is primitive recursive.

(iii) (It is simpler to prove (iii) before (ii))

$\Longrightarrow$: Let $f \in \mathcal{P}^{(1)}$, say $f = \phi_n$. Then $\mathrm{gr}(f)$ is r.e., since

$$y = f(x) \equiv (\exists z)\big(T(z,n,x) \mathbin{\&} y = z.1\big).$$

(Compare example 3.2.)

$\Longleftarrow$: Assume $\mathrm{gr}(f) = \{\, (x,y) \mid y = f(x) \,\}$ is r.e. Then there exists a recursive predicate R such that $\mathrm{gr}(f) = \big\{\, (x,y) \mid (\exists z)\big(R(z,x,y)\big) \,\big\}$. Hence

$$f(x) = \big((\mu u)[R(u.0, x, u.1)]\big).1,$$

which shows that f is partial recursive.

(ii) Similarly as in (i) we already know that if f is recursive, then $\mathrm{gr}(f)$ is recursive. If $\mathrm{gr}(f)$ is recursive, then $\mathrm{gr}(f)$ is r.e. by theorem 1.3 (ii). Finally, if $\mathrm{gr}(f)$ is r.e., then f is partial recursive by (iii), hence f is recursive, since f was assumed to be total. □

Next we describe how $\mathcal{AL}$-programs can be used for generating r.e. sets. An advanced construction of an r.e. set A is usually presented by describing a procedure (in English) which contains instructions of the form 'now enumerate x in A.' Similarly, a partial recursive function can be constructed by generating its graph. Such procedures can be implemented in $\mathcal{AL}$, using the instructions enum and def, which were not yet dealt with in chapter 2. Along with these instructions we now introduce the last few remaining syntactical and semantical features of $\mathcal{AL}$, namely:

1. There are new data types $\mathcal{E}(T)$ and $\mathcal{P}(T,T')$ for any given data types T, T'. If X is of type $\mathcal{E}(T)$, then X represents an r.e. set $A \subseteq T$, and if X is of type $\mathcal{P}(T,T')$, then X represents a partial recursive function $f\colon T \to T'$ to be constructed. At every stage during the (usually non-terminating) computation the value of X is an approximation of A or f, namely a finite set or function. The initial value of X is $\emptyset$ or $\lambda x.{\uparrow}$ respectively.

If $T = T_1 \times \cdots \times T_n$, then of course it does not matter whether $f\colon T \to T'$ is regarded as a function of 1 or n arguments, hence we need not use the notation $\mathcal{P}(T_1, \ldots, T_n, T')$.

$\mathcal{E}(\omega^n)$ and $\mathcal{P}(\omega^n, \omega)$ are abbreviated by $\mathcal{E}^{(n)}$ and $\mathcal{P}^{(n)}$ respectively, and once again the superscript (n) may be omitted if it equals 1.

2. If X is of type $\mathcal{E}(T)$, then there may be no assignments to X, but commands of the form $\mathrm{enum}(X, E)$, where E is an expression of type T. Its meaning is that the current value of E is included in the r.e. set to be constructed which is represented by X. The instruction $\mathrm{enum}(X, E)$ changes the current value of X in the same way as the (hypothetical) assignment $X := X \cup \{E\}$.

3. If X is of type $\mathcal{P}(T, T')$, then again there must be no assignment to X, but there may be instructions of the form $\mathrm{def}(X, E_1, E_2)$, which mean: If f is the function represented by X, define $f(E_1)$ to be E_2. However, we never allow such a value to be redefined, that is, to obtain different values in the course of the computation. Therefore we take the convention that $\mathrm{def}(X, E_1, E_2)$ has no effect if $f(E_1)$ already had a value before. Otherwise $\mathrm{def}(X, E_1, E_2)$ also effects the value of X in the obvious way.

4. Again let X be of type $\mathcal{E}(T)$ or $\mathcal{P}(T, T')$. Since the value of X in every configuration is a finite set or function, we may form expressions from X, using set- or function operations respectively. Examples of valid constructions are $Y := \mathrm{dom}(X)$ or **if** $Z \in X$ **then** ....

5. A variable X may also be of type $\big(\mathcal{E}(T)\big)^*$ or $\big(\mathcal{P}(T, T')\big)^*$. In this case the following rules apply. X is initialized to () as usual. If $E \geq L(X)$, then the current value of $X[E]$ is the empty set or function; this is also in accordance with the conventions of section 2.4, since $\emptyset$ or $\lambda x.\!\uparrow$ serve as 0_A for $A = \mathcal{E}(T)$ or $\mathcal{P}(T, T')$ respectively. Now instructions like $\mathrm{enum}(X[I], E)$ or $\mathrm{def}(X[I], E_1, E_2)$ make sense; their effect may include an increase of the value of $L(X)$ to $I + 1$.

Using variables of types $\big(\mathcal{E}(T)\big)^*$ or $\big(\mathcal{P}(T, T')\big)^*$, an $\mathcal{AL}$-program may generate infinitely many r.e. sets and/or partial recursive functions at the same time.

6. In a **for**-statement **for** $I := E_1$ **to** E_2 **do** S we allow E_2 to be ∞; in this case the meaning is: Execute S for $I = E_1, E_1 + 1, \ldots$, without ever stopping. (Since procedures for generating sets or functions usually do not terminate, it is natural to allow this kind of construct for never-terminating loops.)

Definition 3.6: (V) Let P be a $\mathcal{AL}$-program, X a (possibly subscripted) variable of P.

(i) Assume that X is of type $\mathcal{E}(T)$, $B \subseteq T$. Then P *enumerates (generates)* B *with respect to* X, iff B is the set of all x such that an instruction of the form $\mathrm{enum}(X, E)$ is executed in the computation of P, with the current value of E being x.

(ii) Assume that X is of type $\mathcal{P}(T, T')$, $f\colon T \to T'$. Then P *generates* f *with respect to* X, iff for every $\vec{x}_n$ the following is true: An instruction of the form $\mathrm{def}(X, E_1, E_2)$, with $\vec{x}_n$ being the current value of E_1, is executed in the computation of P iff $f(\vec{x}_n)\!\downarrow$, and in this case the current value of E_2 is $f(\vec{x}_n)$ for the first such instruction to be executed.

Theorem 3.14: (i) Theorem 2.15 remains valid under the syntax extensions just specified.

(ii) A set A is r.e. if and only if A is enumerated by some $\mathcal{AL}$-program (with respect to some variable).

(iii) A function f is partial recursive if and only if f is generated by some $\mathcal{AL}$-program (with respect to some variable).

Proof: (i) It suffices to translate a given $\mathcal{AL}$-program P into an equivalent program P' which does not include the features described under 1.–6. above. This is accomplished by applying the following changes:

1. Change every data type $\mathcal{E}(T)$ to $T^{\#}$ and replace every instruction $\mathrm{enum}(X, E)$ by the assignment $X := X \cup \{E\}$. (This modification and the next involve applications of exercises 2.18 (ii), (iii) and 2.20 (i).)

2. Change every data type $\mathcal{P}(T, T')$ to $\mathcal{F}(T, T')$ and replace every instruction $\mathrm{def}(X, E_1, E_2)$ by the conditional **if** $\neg E_1 \in \mathrm{dom}(X)$ **then** $X := X \cup \{(E_1, E_2)\}$.

3. Replace every infinite **for**-loop **for** $I := E_1$ **to** ∞ **do** S by the following sequence of statements:

> $I := E_1$;
> **while true do**
>   **begin** S; $I := I + 1$ **end**

We assume $A \subseteq \omega$ in part (ii), $f\colon \omega \to \omega$ in (ii); the general case is considered in exercise 3.15.

(ii) $\Longrightarrow$: We construct an $\mathcal{AL}$-program P which enumerates A with respect to X, using condition (i) from theorem 3.8. If $A = \emptyset$, then P consists only of the declaration $X\colon \mathcal{E}$. If $A = \mathrm{ran}\, f$, where $f\colon \omega \to \omega$, then P has the form

> $X\colon \mathcal{E}$;
> **for** $I := 0$ **to** ∞ **do**
> $\mathrm{enum}\big(X, f(I)\big)$

$\Longleftarrow$: Let A be enumerated by P with respect to X. An $\mathcal{AL}$-program P' which computes a function f with domain A with respect to new variables Y, Z is obtained from P as follows: An empty statement with label END is attached at the end of the program, and before every statement of the form $\mathrm{enum}(X, E)$ the instruction **if** $E = Y$ **then goto** END is inserted. The program P' simulates P step by step, but when P enumerates the value of Y (input value) in the set A represented by X, then P' terminates.

(iii) $\Longrightarrow$: We construct an $\mathcal{AL}$-program P which generates f with respect to X. If f is nowhere defined, then P contains only the declaration $X : \mathcal{P}$. Otherwise let $g\colon \omega \to \omega$ be a recursive function with range $\mathrm{dom}\, f$. Now P has the form

> $X : \mathcal{P}$;
> **for** $I := 0$ **to** ∞ **do** $\mathrm{def}\big(X, g(I), fg(I)\big)$.

$\Longleftarrow$: Let P be a $\mathcal{AL}$-program which generates f with respect to X. A program P' which computes f with respect to Y and Z is obtained from P similarly as in the proof of (ii): An empty statement with label END is attached at the end of the program, and before every statement of the form $\mathrm{def}(X, E_1, E_2)$ the instruction **if** $E = Y$ **then begin** $Z := E_2$; **goto** END is inserted. □

Exercise 3.15: [3] Show that theorem 3.14 also holds if $A \subseteq T$ or $f\colon T \to T'$ for arbitrary types T, T'.

Exercise 3.16: [1] Write an $\mathcal{AL}$-program which enumerates the set of all non-Fermat-exponents (exercise 2.10 (ii)) without assuming the result by Wiles and Taylor.

We conclude this chapter by proving the first undecidability results. The following two sets will play an important rôle later on:

Notation: The sets K_0 and K are defined as follows:

$$K_0 = \{ (x, y) \mid \phi_x(y)\downarrow \},$$
$$K = \{ x \mid \phi_x(x)\downarrow \}.$$

Theorem 3.15: The sets K_0 and K are r.e., but not recursive.

Proof: The claim that K_0 and K are r.e. follows from the following representations:

$$(x, y) \in K_0 \iff (\exists z)\big(T(z, x, y)\big),$$
$$x \in K \iff (\exists z)\big(T(z, x, x)\big),$$

If K_0 were recursive, then $K = \{ x \mid (x, x) \in K_0 \}$ would also be recursive. We now derive a contradiction from the latter assumption. If K were recursive, then $\overline{K}$ would be r.e. by theorem 3.11, hence $\overline{K} = W_u$ for some u. But this implies

$$\begin{aligned} u \in K &\iff u \in W_u \quad \text{(by definition of } K\text{)} \\ &\iff u \in \overline{K} \quad (\text{since } \overline{K} = W_u), \end{aligned}$$

which is a impossible. □

The last part of this proof, where it was shown that $\overline{K}$ is not r.e., may appear somewhat mysterious at first glance. Actually, when looked at from the right angle, this is yet another application of diagonalization. Let us take the position that the set K is not given, but that we wish to construct a set $A \subset \omega$ which is not r.e. Then A must differ from each of the sets W_x, so we define A such that A differs from W_x at x, that is,

$$(\forall x)(x \in A \iff x \notin W_x),$$
$$A = \{ x \mid x \notin W_x \}.$$

In this way A becomes exactly the set $\overline{K}$.

Definition 3.7: Let A be a data type, $B \subseteq A$. If B is not recursive, then the problem of deciding whether or not $\vec{x}_n \in B$, as well as B itself, is called *undecidable* (*algorithmically undecidable*, *recursively undecidable*).

Definition 3.8: The problem of deciding for a given input $(x, \vec{y}_n)$ if $\phi_x^{(n)}(\vec{y}_n)\downarrow$ is called the *halting problem*.

Thus the halting problem is the first problem shown to be undecidable. Theorem 3.15 only dealt with the case $n = 1$, but as usual the proof carries over to $n > 1$.

Exercise 3.17: [2] Let $f\colon \omega \to \omega$ be a one-one recursive function. Show that the set $A = \{ x \mid f(x) \in W_x \}$ is r.e., but not recursive. Is the same also true for the set $B = \{ x \mid x \in W_{f(x)} \}$?

Exercise 3.18: [2] Let $R(\vec{x}_m, \vec{y}_n)$ be a recursive predicate, and let

$$A = \{\, \vec{y}_n \mid (\forall \vec{x}_m)\big(R(\vec{x}_m, \vec{y}_n)\big) \,\}.$$

Show that $\overline{A}$, but not necessarily A, is r.e.

Exercise 3.19: [4] The purpose of this exercise is to show that in theorem 3.2 the application of the function d is necessary, and to characterize exactly which functions could be used in the place of d, provided that $T^{(n)}$ is replaced by a suitable predicate.

(i) Show that there exists a partial recursive function $f\colon \omega \to \omega$ which is not of the form $\lambda x.(\mu z)[P(z,x)]$ for any recursive predicate P. *Hint:* Construct f such that the assumption $f = \lambda x.(\mu z)[P(z,x)]$ with P recursive would imply that K is recursive.

(ii) Let $d\colon \omega \to \omega$, and assume that for some b there exist only finitely many x such that $d(x) = b$. Show that there exists a partial recursive function $f\colon \omega \to \omega$ which is not of the form $\lambda x.d\big((\mu z)[P(z,x)]\big)$ for any recursive predicate P. Thus the function d from theorem 3.3 must take every value infinitely often. (This generalizes (i), since (i) is obtained by taking $d = \lambda x.x$.)

(iii) Let d be as in (ii). Show that there exists a *recursive* function $f\colon \omega \to \omega$ which is not of the form $\lambda x.d\big((\mu z)[P(z,x)]\big)$ for any *primitive recursive* predicate P.

(iv) Let $d\colon \omega \to \omega$ be (primitive) recursive such that d takes every value infinitely often. Prove that for every n there exists a (primitive) recursive predicate P such that for all u

$$\phi_u^{(n)}(\vec{x}_n) = d\big((\mu z)[P(z,u,\vec{x}_n)]\big).$$

3.4. Church's thesis.

ChT states essentially that all procedures (in the intuitive sense) can be represented in $\mathcal{WL}$ (or in any equivalent formalism for defining the class of partial recursive functions, like those of chapter 4). More precisely, it says:

(i) A partial function $f\colon \omega^n \to \omega$ is partial recursive if and only if it is computed by some procedure, as explained in section 1.1.

(ii) A set $A \subseteq \omega^n$ is recursive if and only if there exists an algorithm which decides for given $\vec{x}_n$ whether or not $\vec{x}_n \in A$.

(iii) A set $A \subseteq \omega^n$ is recursively enumerable if and only if A is generated (or accepted) by some procedure.

This thesis is of course named after Alonzo Church, who was one of the founders of the theory of recursive functions in the 1930's, along with A.M. Turing, K. Gödel, S.C. Kleene and B.J. Rosser. Naturally, it can only be a thesis, not a mathematical theorem, since the notion of procedure is not a mathematically precise concept. In order to convert it into a precise mathematical statement, one would have to replace the notion of procedure by some mathematical equivalent; but then the question whether or not it is really equivalent is exactly what ChT is about.

In the following we will only discuss assertion (i) of ChT; the other two parts certainly have the same status of plausibility. Note first that definition 1.1 requires that a procedure can be executed systematically and mechanically; if there were some superman who is capable of telling $f(\vec{x}_n)$ for any given $\vec{x}_n$ thanks to some ingenious intuition, than it would be wrong to say that he follows an algorithm. On the other hand, an algorithm is not necessarily something that can be performed *in practice*, but only under idealized circumstances, that is, if an unlimited amount of time and space were available, and if no miscalculations were made.

It was already noted in section 1.1 that one direction of ChT clear: Every partial recursive function is indeed computable by a procedure, since $\mathcal{WL}$-programs satisfy all requirements of definition 1.1. However, the converse of this assertion is by no means obvious. To a considerable extend it is supported by experience: Nobody has been able to exhibit a 'computable' function which is not partial recursive, and in fact there is not even the slightest hint as to how such a function could be constructed. Such a 'counterexample' might either be an ad hoc algorithm for some specific problem, or a new model of computation which is more powerful than those considered so far. As will be seen in chapter 4, numerous attempts to define a mathematical equivalent for the class of 'computable' functions all turned out to be equivalent. The property of $\mathcal{P}$ that it is invariant under many modifications of its definition is called *absoluteness*. It indicates at the very least that $\mathcal{P}$ is a very interesting class of function which is worth being investigated further. Still, all this provides only *empirical* support for ChT.

Another line of argumentation consists in analyzing what exactly is going on when either a human or a computing machine (not necessarily a computer as it is known today) carries out an algorithm. Such an analysis is given, for instance, in Turing [26], and, along different lines, in McNaughthon [15], p. 162 ff. Nevertheless, all these arguments are based on certain assumptions which, though highly plausible, leave some room for disagreement. Overall, ChT is widely accepted as true today, and most mathematicians regard the evidence in favor of it as very strong, if not overwhelming; but an absolutely compelling argument, that would put an end to all discussion, is not in sight.

The rôle played by ChT is twofold. Many results in the theory of recursive functions state that a certain function f is not partial recursive. In this form such an assertion has an unambiguous, purely mathematical meaning. However, if one accepts ChT, then a more drastic conclusion can be made, namely that there cannot be a systematic method for computing $f(\vec{x}_n)$ for any given $\vec{x}_n$.

Secondly, in order to show that a partial recursive function f with certain properties exists, one frequently provides an algorithm in the intuitive sense for computing f. This procedure is commonly referred to as a *proof by ChT*. Actually, ChT is not really essential for such a proof. Instead, it just follows a pattern which is quite common in mathematics: Details of proofs which are straightforward but tedious and not very informative are typically omitted. In general experienced mathematicians simply know where such shortcuts may be applied (though occasionally mistakes have occurred). Of course one has to be prepared to supply missing details when challenged.

Assume that a method for computing f has been described informally. Then an $\mathcal{AL}$-program (or some other representation which proves partial recursiveness) is just an example of the kind of detail which may be omitted in a mathematical proof. Indeed, given an informal description of an algorithm $\mathcal{A}$, an experienced programmer usually knows that $\mathcal{A}$ can be implemented, long before such an implementation has been performed. However, this knowledge depends on his/her acquaintance with the power of the programming language to be used. For instance, nobody can be expected to realize that the Euclidean algorithm can be implemented in $\mathcal{WL}$ immediately after learning the definition of $\mathcal{WL}$, without knowledge of some of the results from chapter 2.

Exercise 3.20: [2] In each of the parts (i)–(iv) determine whether or not an algorithm for computing a function f is described, so that by ChT one can conclude that f is partial recursive. The point is not to decide whether or not these instructions are precise, but whether or not they can be carried out effectively.

(i) Given x, start enumerating (generating) W_x, and let $f(x)$ be the second element to appear in this enumeration. If W_x contains less than two elements, then $f(x)$ should be undefined.

(ii) Given x, start enumerating W_x. If at least two elements appear in this enumeration, then let $f(x)$ be the smaller of the first two elements to appear. If only one element appears, then let $f(x)$ be that element. If no element appears, then let $f(x)$ be undefined.

(iii) Given x and y, try to compute $\phi_x(x)$ and $\phi_y(y)$. If the computation of $\phi_x(x)$ terminates, then let $f(x) = \phi_x(x)$, otherwise let $f(x) = \phi_y(y)$ (which includes the case where $\phi_y(y)$ is also undefined).

(iv) Given x, try to compute $\phi_x(0)$; if this computation does not terminate, then $f(x)\uparrow$. If $\phi_x(0)\downarrow$, say $\phi_x(0) = a$, then check for $p = a$, $a + 1$, $\ldots$, whether or not p and $p+x$ are two consecutive prime numbers. Then let $f(x)$ be the smallest p for which this is the case. If no such p exists, then let $f(x)\uparrow$.

In texts on recursive function theory, especially in more advanced treatments, proofs by ChT are generally regarded as acceptable, and ChT is in fact used very extensively. This means that the gap between the given presentation of proofs and strict formal proofs (with no details omitted) is relatively great. There is nothing wrong with this, and it does not at all mean that such proofs are unreliable. Nevertheless, in these lectures notes we have attempted to bridge the gap to some extend, by providing results which allow to formalize proofs by ChT. For instance, theorem 3.14 allows to formalize proofs which involve the generation of an r.e. set by a procedure.

Up to now all proofs in these lecture notes were carried out without an appeal to ChT. In the chapters to follow proofs will still be made formal whenever this can be done without an excessive amount of work. In the exercises, too, the reader is still asked to avoid the use of ChT, unless it is stated otherwise.

4. Alternative models of computation.

In this chapter some alternative formalisms are discussed by which the class of partial recursive functions could be defined. A considerable number of such formalisms have been considered in literature, and all have been shown to be equivalent. Naturally, a complete treatment would be beyond the frame of these lecture notes. Instead, it will only be shown that the following three formalisms are equivalent to that of $\mathcal{WL}$-programs: *μ-recursion*, *Turing machines* and *Markov algorithms*. Other formalisms will be mentioned briefly at the end of the chapter.

The present chapter has to be regarded as a digression, if we take the point of view of pure recursive function theory, where we are interested only in the structure of $\mathcal{P}$, $\mathcal{E}$ etc., not in applications for undecidability proofs outside this area, or in complexity issues. In chapter 5 we will continue proving results of the kind which were typical for chapter 3. The results of the present chapter are important for two reasons: They provide additional support for ChT, as explained in section 3.4, and some of them will be needed in (DCC).

4.1. μ-recursion.

The formalism of μ-recursion is due to Kleene [9], [10]. It differs from most others in so far, as no class of algorithms is defined, but only a class of 'computable' functions. These are those functions which can be obtained from certain very simple initial functions by means of certain operations.

Definition 4.1: The following functions are the *initial functions*:

$$\begin{aligned} \zeta &= \lambda .0 && \text{((0-place) zero function),} \\ s &= \lambda x.x+1 && \text{(successor function,)} \\ u_i^n &= \lambda \vec{x}_n.x_i && \text{(projection functions).} \end{aligned}$$

The set of all initial functions is denoted by I; thus

$$I = \{\zeta, s\} \cup \{\, u_i^n \mid 1 \le i \le n \,\}.$$

Three operations for constructing new functions from old ones are involved in the formalism of μ-recursion. One, namely primitive recursion, has already been defined (definition 2.6 (i)); the other two are a special case and a variant of definition schemes considered earlier.

Definition 4.2: (i) A function $h(\vec{x}_n)$ is defined from functions $f(\vec{x}_m)$, $g_1(\vec{x}_n)$, ..., $g_m(\vec{x}_n)$ by *composition*, iff $h(\vec{x}_n) = f\big(g_1(\vec{x}_n), \dots, g_m(\vec{x}_n)\big)$.

(ii) A function $g(\vec{y}_n)$ is defined from $f(x, \vec{y}_n)$ by (*(unbounded) minimization*, iff

$$g(\vec{y}_n) = \begin{cases} \text{the smallest } x \text{ such that } f(x,\vec{y}_n) = 0 & \\ \ \& \ f(z,\vec{y}_n)\!\downarrow \text{ for all } z < x & \text{if such an } x \text{ exists,} \\ \uparrow & \text{if no such } x \text{ exists.} \end{cases}$$

It will be seen in the proof of theorem 4.1 that it suffices to apply the operation of minimization to *total* functions. Under this restriction definition 4.2 (ii) can be simplified by defining $g = \lambda\vec{y}_n.(\mu z)[f(z, \vec{y}_n) = 0]$ (see definition 2.7 (ii)). As in the proof of theorem 2.7 (i) we have the following obvious procedure for computing $g(\vec{y}_n)$, assuming that f is algorithmically computable: Compute $f(0, \vec{y}_n)$, $f(1, \vec{y}_n)$, $\ldots$, until a z has been found such that $f(z, \vec{y}_n) = 0$; then output z. However, if f is not total, then this procedure computes the function g as defined in definition 4.2 (ii), not the function $\lambda\vec{y}_n.(\mu z)[f(z, \vec{y}_n) = 0]$. Indeed, if z is minimal such that $f(z, \vec{y}_n) = 0$, and $f(z', \vec{y}_n)\uparrow$ for some $z' < z$, then this procedure will get stuck in the computation of $f(z', \vec{y}_n)$ and yield no output. Nor does there exist any other procedure for computing $\lambda\vec{y}_n.(\mu z)[f(z, \vec{y}_n) = 0]$ in general, as the following exercise shows:

Exercise 4.1: [3] Show that there exists a partial recursive function $f(x, y)$ such that $g = \lambda y.(\mu z)[f(z, y) = 0]$ is not partial recursive. *Hint:* Construct f such that the assumption that g is partial recursive would imply that K is recursive.

The following is the main result of this section:

Theorem 4.1: (i) A function f is primitive recursive if and only if it can be defined from initial functions by (repeated application of) composition and primitive recursion.

(ii) A function f is partial recursive if and only if it can be defined from initial functions by (repeated application of) composition, primitive recursion and minimization.

For the rest of this section let $\mathrm{PR}(f)$ ($\mathrm{P}(f)$) be the assertion that f can be defined from initial functions by composition, primitive recursion (and minimization). Its precise meaning is that there exists a sequence of functions f_0, $\ldots$, f_{l-1} with $f_{l-1} = f$ such that each f_i is either an initial function, or f_i is defined from some of the functions f_j, $j < i$, by composition or primitive recursion (or minimization). Such a sequence of functions is called a *derivation of* f as a primitive (partial) recursive function. Naturally, this terminology makes sense only if theorem 4.1 is already assumed, or if the predicates PR and P are used to *define* the terms *primitive recursive* and *partial recursive*, as it is done in many textbooks.

In both parts of theorem 4.1 the *if*-part follows almost immediately from the results of chapter 2. By contrast, the *only if*-part still requires a certain amount of work; in particular, we have to show $\mathrm{PR}(f)$ for a certain collection of functions f. Hence the following examples and exercises serve not only as an illustration, but also as a preparation for the proof of theorem 4.1.

Example 4.1: In each of the following parts we show that the function f under consideration satisfies $\mathrm{PR}(f)$:

(i) The function $f_1 = \lambda xy.x + y$. It can be defined by primitive recursion, similarly as it is done when addition is defined axiomatically:

$$\begin{aligned} f_1(0, y) &= y = u_1^1(y), \\ f_1(x+1, y) &= s\big(f_1(x, y)\big) = s\big(u_3^3(x, y, f_1(x, y))\big). \end{aligned}$$

The expression $s\big(u_3^3(x, y, f_1(x, y))\big)$ has to be used at the right side in order to adjust this definition to the scheme of primitive recursion. This expression defines a function h of x, y and $f_1(x, y)$, which is obtained from s and u_3^3 by composition, so that $\mathrm{PR}(h)$ holds.

(ii) The function $f_2 = \lambda xy.xy$ can also be defined by primitive recursion using (i), though it will be seen that this involves a minor difficulty. We have

$$\begin{aligned} f_2(0,y) &= 0, \\ f_2(x+1,y) &= y + f_2(x,y) = f_1\big(u_2^3(x,y,f_2(x,y)),u_3^3(x,y,f_2(x,y))\big). \end{aligned}$$

Here $f_1\big(u_2^3(x,y,f_2(x,y)),u_3^3(x,y,f_2(x,y))\big)$ represents a a function of x, y and $f_2(x,y)$ obtained from f_1, u_2^3 and u_3^3 by composition. However, the right side of the first equation, 0, has to be regarded as a function of y, so we still have to show that $\lambda y.0$ satisfies PR. This is the next example; as soon as it is completed, example (ii) will be finished, too.

(iii) (P) $\zeta^{(1)} = \lambda x.0$ is obtained as follows:

$$\begin{aligned} \zeta^{(1)}(0) &= \zeta(), \\ \zeta^{(1)}(x+1) &= \zeta^{(1)}(x) = u_2^2\big(x,\zeta^{(1)}(x)\big). \end{aligned}$$

This is indeed an instance of primitive recursion, where the number n from definition 2.4 (ii) is 0.

(iv) $f_3 = \lambda x.x \dot{-} 1$ is obtained as follows:

$$\begin{aligned} f_3(0) &= \zeta(), \\ f_3(x+1) &= x = u_1^2\big(x,f_3(x)\big). \end{aligned}$$

(v) Finally, we consider function $f_4 = \lambda xy.x \dot{-} y$; its construction demonstrates a new method for using the projection functions to manipulate arguments. The obvious approach is to define f_4 by primitive recursion from f_3, and indeed

$$\begin{aligned} f_4(x,0) &= x, \\ f_4(x,y+1) &= f_3\big(f_4(x,y)\big). \end{aligned}$$

However, this definition does not yet concur with the scheme of primitive recursion, which allows only recursion on the first variable. This difficulty is resolved by first defining $f_5 = \lambda xy.y \dot{-} x$:

$$\begin{aligned} f_5(0,y) &= y, \\ f_5(x+1,y) &= f_3\big(f_5(x,y)\big) = f_3\big(u_3^3(x,y,f_5(x,y))\big). \end{aligned}$$

Then f_4 is obtained by composition, using the projection functions to swap arguments:

$$f_4(x,y) = f_5\big(u_2^2(x,y),u_1^2(x,y)\big).$$

Exercise 4.2: Show that the following functions f satisfy PR(f):

(i) [1] $\lambda x.x!$.

(ii) [2] $\lambda xy.x^y$.

(iii) [2] $\lambda x.\operatorname{sig} x$, see exercise 1.4 (iv).

(iv) [4] $\lambda xy.\lfloor x/y \rfloor$, see exercise 2.2 (ii) for this and the following part.

(v) [1] $\lambda xy.x \bmod y$.

Exercise 4.3: Show that the following functions f satisfy $\mathrm{P}(f)$:

(i) [0] For every n the nowhere defined function (P) $\lambda\vec{x}_n.\uparrow$.

(ii) [4] The function f defined by

$$f(x,y) = \begin{cases} x-y & \text{if } x \geq y, \\ \uparrow & \text{if } x < y. \end{cases}$$

In order to facilitate our further task, we now prove analogues of some of the results from chapter 2 for the formalism of (μ)-recursion:

Lemma 4.2: Let $\mathcal{F} \subseteq \mathcal{NF}$ be a set of functions such that $\mathrm{PR}(f)$ holds for all $f \in \mathcal{F}$, $\mathcal{V} = \{x_1, \ldots, x_n\}$, and let t be a numerical $(\mathcal{V}, \mathcal{F})$-expression. Then $\mathrm{PR}(g)$ holds for the function $g = \lambda\vec{x}_n.t$.

Proof: As in the proof of theorem 2.4 we proceed by induction on the length of t. If t is a variable, say x_i, then $g = u_i^n$. Next, let t be the constant 0. The function $\zeta^{(n)} = \lambda\vec{x}_n.0$ equals ζ for $n = 0$, and in example 4.1 (iii) it was shown that $\mathrm{PR}(\zeta^{(1)})$ holds. If $n > 1$, then $\zeta^{(n)}$ is obtained by composition: $\zeta^{(n)}(\vec{x}_n) = \zeta^{(1)}\big(u_1^n(\vec{x}_n)\big)$. If t is a constant $c > 0$, then $g = \lambda\vec{x}_n.c$ is obtained by composing $\zeta^{(n)}$ with s c times:

$$g(\vec{x}_n) = \underbrace{s(s(\ldots s}_{c}(\zeta^{(n)}(\vec{x}_n))\ldots)).$$

Finally, let $t = f(t_1, \ldots, t_m)$ with $f \in \mathcal{F}$, and assume that the assertion holds for all t_i. Then g is obtained by composition from f, g_1, …, g_m, where $g_i = \lambda\vec{x}_n.t_i$. Since $\mathrm{PR}(g_i)$ holds for all i, $\mathrm{Pr}(g)$ holds, too.

Exercise 4.4: Show that $\mathrm{PR}(f)$ holds for the functions f from theorem 2.8, namely:

(i) [1] The functions ι_n, $n \geq 1$.

(ii) [4] The functions κ_i^n, $1 \leq i \leq n$.

Proof of theorem 4.1: We first show the direction $\Longleftarrow$ for both parts. This amounts to establishing the following facts:

(a) All initial functions are primitive recursive.

(b) The sets $\mathcal{PR}$ and $\mathcal{P}$ are closed under composition and primitive recursion.

(c) $\mathcal{P}$ is also closed under minimization.

Fact (a) is trivial, and (b) follows from theorems 2.4 (i) and 2.6. Now assume that $g(\vec{y}_n)$ is defined from $f(x, \vec{y}_n)$ by minimization, where f is partial recursive. To show that g is partial recursive, the proof of theorem 2.7 (i) has to be slightly modified, since the predicate $\lambda x\vec{y}_n.f(x,\vec{y}_n) = 0$ is not necessarily recursive, see exercise 4.1. But the following program, which avoids a nonrecursive **while**-condition, computes g with respect to Y_1, …, Y_n, Z:

$Z := 0$; $X := f(0, Y_1, \ldots, Y_n)$;
while $X \neq 0$ **do**
  begin $Z := Z + 1$; $X := f(Z, Y_1, \ldots, Y_n)$ **end**

This establishes (c).

Next, assume that the direction $\Longrightarrow$ has already been shown for part (i); we prove it for (ii). We have to show that for every u and n the assertion $\mathrm{P}(\phi_u^{(n)})$ holds. Let f be the characteristic function of $\lambda z\vec{x}_n.\neg T^{(n)}(z,u,\vec{x}_n)$. Then

$$\phi_u^{(n)}(\vec{x}_n) = d\big((\mu z)[T^{(n)}(z,u,\vec{x}_n)]\big) = d\big((\mu z)[f(z,\vec{x}_n) = 0]\big).$$

Since $\mathrm{PR}(f)$ and $\mathrm{PR}(d)$ are true by assumption, we are done. Moreover, it follows that the operator of minimization has to be applied only once, and only to a primitive recursive function. It remains to show $\Longrightarrow$ for part (i); this is the main part of the proof.

Let $f(\vec{x}_n)$ be a primitive recursive function, computed by a special $\mathcal{WL}$-program P with respect to $X_1, \ldots, X_n, X_{n+1}$. Also, let $R = \{X_1, \ldots, X_m\}$ be the set of variables of P, where $m \geq n+1$, and let $U = (R, P)$. In what follows we assume the notation introduced in and before theorem 1.1; in particular, we use the symbols $\mathcal{V}$ and $\Phi_{U'} = \Phi_{P'}$ for any URM $U' = (R, P')$ whose set of registers is R. With every $V \in \mathcal{V}$ we associate the code number $\mathrm{cd}_{\mathcal{V}}(V) = \iota_m(\vec{x}_m)$, where $x_i = V(X_i)$. Now the function $(\Phi_{P'})_\omega$ is well-defined for every $\mathcal{WL}$-program P' whose set of variables is contained in R; we denote it by $f_{P'}$. Thus if $\Phi_{P'}(V) = V'$, $V(X_i) = x_i$, $V'(X_i) = x_i'$ for $i = 1, \ldots, m$, then

$$f_{P'}\big(\iota_m(\vec{x}_m)\big) = \iota_m(\vec{x}_m').$$

Let V and V' be the initial and the final configuration of U for input $\vec{x}_n$. Then $\mathrm{cd}_{\mathcal{V}}(V) = \iota_m(x_1, \ldots, x_n, 0, \ldots, 0)$, and the result of the computation is

$$f(\vec{x}_n) = V'(X_{n+1}) = \kappa_{n+1}^m\big(\mathrm{cd}_{\mathcal{V}}(V')\big) = \kappa_{n+1}^m\big(f_P(\iota_m(x_1, \ldots, x_n, 0, \ldots, 0))\big).$$

Hence $\mathrm{PR}(f)$ follows from $\mathrm{PR}(f_P)$. We prove the latter assertion for every special $\mathcal{WL}$-program P whose set of variables is contained in R, using induction on the length of P. This involves an application of exercise 1.1 and theorem 1.1.

(a) If P is empty, then $f_P = u_1^1$.

(b) If P consists of the single statement $X_i := X_i + 1$, then

$$f_P(x) = \iota_m\big(\kappa_1^m(x), \ldots, \kappa_{i-1}^m(x), \kappa_i^m(x) + 1, \kappa_{i+1}^m(x), \ldots, \kappa_m^m(x)\big).$$

Similarly, if P consists of the statement $X_i := X_i \dot{-} 1$, then

$$f_P(x) = \iota_m\big(\kappa_1^m(x), \ldots, \kappa_{i-1}^m(x), \kappa_i^m(x) \dot{-} 1, \kappa_{i+1}^m(x), \ldots, \kappa_m^m(x)\big).$$

(c) If P is of the form $P_1; P_2$, then $f_P = f_{P_2} f_{P_1}$, hence $\mathrm{PR}(f_P)$ holds by induction hypothesis.

(d) Finally, assume that P is of the form **while** $X_i > 0$; P_1; **end**, where P_1 begins with the assignment $X_i := X_i \dot{-} 1$ and contains no other assignments to X_i. Then the **while**-loop is executed x_i times, where x_i is the initial value of X_i. Similarly as in the proof of theorem 1.1 we have

$$\Phi_P(x_1, \ldots, x_m) = \Phi_{P_1}^{x_i}(x_1, \ldots, x_m),$$

which implies

$$f_P(x) = f_{P_1}^{\kappa_i^m(x)}.$$

Hence $\mathrm{PR}(f_P)$ holds by induction hypothesis, since the function $\lambda yx.f_{P_1}^y(x)$, which is obtained from f_{P_1} by iteration, can be defined by primitive recursion as in example 2.4 (vi). This completes the proof of the theorem. □

4.2. The general machine concept.

In section 4.3 and 4.4 two more classes of computing machines will be defined, and in each case the definition of the function computed by such a machine follows a similar scheme as definition 1.3: The input determines an initial configuration, then the computation proceeds by repeatedly transforming one configuration into the next, until a halting configuration is reached, and this final configuration determines the result. This suggests the following general

Definition 4.3: (V) (i) A (computing) machine (automaton) M is a septuple of the form $(I_M, O_M, \mathcal{CF}_M, \mathrm{halt}_M, \mathrm{init}_M, \mathrm{next}_M, \mathrm{res}_M)$ such that

(a) I_M, O_M and $\mathcal{CF}_M$ are countable sets for which coding functions cd_{I_M}, cd_{O_M} and $\mathrm{cd}_{\mathcal{CF}_M}$ are defined. They are called the *input domain*, *output domain* and the *set of configurations* of M.

(b) halt_M is a recursive predicate on $\mathcal{CF}_M$; if $\mathrm{halt}(\mathcal{C})$ holds for $\mathcal{C} \in \mathcal{CF}$, then $\mathcal{C}$ is called a *halting configuration*.

(c) The functions $\mathrm{init}_M: I_M \to \mathcal{CF}_M$, $\mathrm{next}_M: \mathcal{CF}_M \to \mathcal{CF}_M$ and $\mathrm{res}: \mathcal{CF}_M \to O_M$ are partial recursive; they are called *input-*, *transition-* and *output function.* In fact, init_M has to be recursive, $\mathrm{next}_M(\mathcal{C})$ has to be defined at least for all non-halting configurations $\mathcal{C}$, and res at least for all halting configurations $\mathcal{C}$. If $\mathrm{next}(\mathcal{C}) = \mathcal{C}'$, then we say that $\mathcal{C}$ *yields* $\mathcal{C}'$ (in one step of a computation by M), and we also write $\mathcal{C} \underset{M}{\longrightarrow} \mathcal{C}'$.

(ii) The *computation of* M *starting in the configuration* $\mathcal{C}$ is the (uniquely determined) sequence of configurations $\Gamma = (\mathcal{C}_0 = \mathcal{C}, \mathcal{C}_1, \ldots)$ such that $\mathrm{next}(\mathcal{C}_{i-1}) = \mathcal{C}_i$ for $i = 1$, $2, \ldots$, and such that Γ ends with the configuration $\mathcal{C}_t$ if and only if $\mathcal{C}_t$ is a halting configuration. The terms *initial configuration*, *final configuration* and *step* of Γ are defined in the obvious way, analoguously as in definition 1.4.

(iii) For $x \in I_M$ the computation of M for input x is the computation starting in the configuration $\mathrm{init}_M(x)$. The function $f: I_M \to O_M$ computed by M is defined as follows: For all $x \in I_M$, $f(x){\downarrow}$ if and only if the computation of M for input x terminates, and if it terminates in the configuration $\mathcal{C}'$, then $f(x) = \mathrm{res}_M(\mathcal{C}')$.

Note that if M computes f, then M represents an algorithm for computing f in the sense of definition 1.1. Examples of textbooks where the notion of machine is also introduced are Brainerd/Landweber [2], p. 86/87 and Weihrauch [27], p. 19. Their definitions differ from definition 4.3, but not in an essential way.

We will later refer to URM's as examples of machines, although, strictly speaking, this is not correct: The components of a URM (program and set of registers) differ as mathematical objects from those of machines. However, a machine M computing f can be constructed from a URM U computing f in a straigtforward way, using definition 1.4.

Among the components of M, init_M and res_M depend not only on U, but also on the set of input- and output variables chosen. Such a construction is asked for in exercise 4.5.

Analogous remarks also apply for Turing machines and Markov algorithms, which are introduced in the next two sections. Both will be referred to as machines, though strictly speaking they only *correspond* to machines in an obvious way. For Markov algorithms the conversion into machines will be presented in detail in the second part of the proof of theorem 4.5.

Theorem 4.3: Let $f: A \to B$, where A and B are countable sets for which coding schemes have been defined. Then f is computed by some machine M if and only if f is partial recursive. (This is yet another example of a result where it suffices to consider functions of one argument, since A may also be of the form $A_1 \times \cdots \times A_n$; compare the remark before the proof of theorem 2.10.)

Proof: $\Longrightarrow$: This follows similarly as in the proof of theorem 3.2. Let f be computed by M, with init_M etc. as in definition 4.3. Then f is computed by the following $\mathcal{AL}$-program:

$X{:}\, I_M$; $Y{:}\, O_M$; $C{:}\, \mathcal{CF}$;
$C := \mathrm{init}_M(X)$;
while $\neg\mathrm{halt}_M(C)$ **do** $C := \mathrm{next}_M(C)$;
$Y := \mathrm{res}(C)$

$\Longleftarrow$: Let f be partial recursive, and let $U = (R, P)$ be a URM computing f_ω with respect to X_1 and X_2, where $R = \{X_1, \ldots, X_m\}$ and $P = I_0; \ldots; I_{l-1}$. A machine M computing f is obtained from U in a straightforward way, see the following exercise. □

Exercise 4.5: [2] Complete the proof of theorem 4.3 by defining the machine M in detail.

The *complexity* of a computation by a machine M can be defined in a very natural way: it is the number of steps involved (definition 4.3 (ii)). If the complexity of the computation for input x is denoted by $\Phi_M(x)$, then the complexity of M itself (or of the procedure represented by M) is the function $\lambda x.\Phi_M(x)$. This notion is of particular importance in the case where M is a Turing machine, and it will be dealt with further in (DCC).

Several theorems in these lecture notes are of the following form: If M is a machine with certain properties computing f, then there also exists a machine with certain other properties computing f. For instance, theorems 2.1–2.3 can be viewed that way. The proof usually consists in constructing M' such that M' *simulates* M. Informally, this means that every configuration $\mathcal{C}$ of M corresponds to certain configurations $\mathcal{C}'$ of M', where $\mathcal{C}'$ may contain more information than $\mathcal{C}$, and that every step of a computation by M corresponds to a sequence of steps of a computation by M' (compare Knuth [11], exercise 1.1.9). In the following formal definition $\mathcal{C} \in \mathcal{CF}_M$ corresponds to $\mathcal{C}' \in \mathcal{CF}_{M'}$ iff $h(\mathcal{C}') = \mathcal{C}$.

Definition 4.4: (V) Consider machines

$$M = (I_M, O_M, \mathcal{CF}_M, \mathrm{halt}_M, \mathrm{init}_M, \mathrm{next}_M, \mathrm{res}_M),$$
$$M' = (I_{M'}, O_{M'}, \mathcal{CF}_{M'}, \mathrm{halt}_{M'}, \mathrm{init}_{M'}, \mathrm{next}_{M'}, \mathrm{res}_{M'}).$$

M' *simulates* M, iff

(i) $I_{M'} = I_M$ and $O_{M'} = O_M$,

and there exists an *onto* function $h: \mathcal{CF}_{M'} \to \mathcal{CF}_M$ such that

(ii) For every $x \in I_M$ there exists a configuration $\mathcal{C}' \in \mathcal{CF}_{M'}$ such that $\mathrm{init}_{M'}(x) \xrightarrow[M']{*} \mathcal{C}'$ and $h(\mathcal{C}') = \mathrm{init}_M(x)$. (Hence if Γ and Γ' are the computations of M and M' for input x, then some configuration of Γ' corresponds to the initial configuration of Γ'.)

(iii) If $\mathcal{C}_1, \mathcal{C}_2 \in \mathcal{CF}_M$, $\mathcal{C}_1 \xrightarrow[M]{} \mathcal{C}_2$ and $\mathcal{C}_1 = h(\mathcal{C}'_1)$, then there exists $\mathcal{C}'_2 \in \mathcal{CF}_{M'}$ such that $\mathcal{C}'_1 \xrightarrow[M']{*} \mathcal{C}'_2$ and $h(\mathcal{C}'_2) = \mathcal{C}_2$. (Thus every step of a computation by M is simulated by finitely many steps of a computation by M'.)

(iv) If $\mathcal{C}$ is a halting configuration of M and $\mathcal{C} = h(\mathcal{C}')$, then there exists a halting configuration $\mathcal{C}'_1$ of M' such that $\mathcal{C}' \xrightarrow[M']{*} \mathcal{C}'_1$ and $\mathrm{res}_M(\mathcal{C}) = \mathrm{res}_{M'}(\mathcal{C}'_1)$.

Exercise 4.6: [2] Prove formally that if M' simulates M, then M and M' compute the same function $f: I_M \to O_M$.

The configurations of M may or may not contain information about M itself. In definition 1.4 they do not, but in section 3.1 the tuple $\hat{\mathcal{C}}$ associated with a configuration $\mathcal{C}$ of a URM $U = (R, P)$ contains P as a component. This led to the construction of machines computing the universal functions $\psi^{(n)}$ in the proof of theorem 3.2.

This method may be generalized. Let $\mathcal{M}$ be a class of machines such that the set of all functions computed by some $M \in \mathcal{M}$ is $\mathcal{P}$. (Now we restrict ourselves again to functions from ω^n to ω.) Also, let $\mathcal{M}^{(n)}$ be the set of all $M \in \mathcal{M}$ computing functions of n arguments. Under appropriate conditions all $M \in \mathcal{M}^{(n)}$ may be combined to construct a *universal machine* $M_U^{(n)}$ computing the univeral function corresponding to some enumeration of $\mathcal{P}^{(n)}$ (see definition 3.1 (ii)). In this way it is possible to construct universal Turing machines or universal Markov algorithms, for instance.

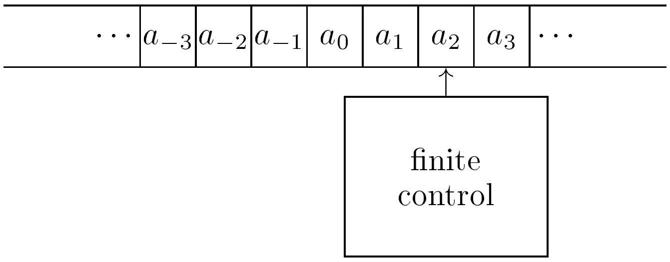
Universal machines $M_U^{(n)}$ were studied in Buchberger/Roider [3]; see also Börger [1], p. 129–136. In particular, criteria are given which specify exactly, which functions and predicates may serve as $\mathrm{halt}_{M_U^{(n)}}$, $\mathrm{init}_{M_U^{(n)}}$, $\mathrm{next}_{M_U^{(n)}}$ or $\mathrm{res}_{M_U^{(n)}}$ respectively.

Exercise 4.7: [2] Show that theorem 3.3 immediately implies the existence of a universal machine M computing $\psi^{(n)}$, that is, construct such M from $T^{(n)}$ and d.

4.3. Turing machines.

Among all models of computation, Turing machines (TM's) play perhaps the most important rôle in theoretical investigations. Like URM's, they are motivated by a conception of some real computing device. In fact, A. Turing [26] analyzed how any mechanical computation proceeds, and he concluded that any such computation can be done by a TM, since on the most basic level the instructions of a computing device are essentially those of a TM.

The type of TM T considered here consists of a finite control, a two-way infinite tape divided into cells, and a reading/writing tape head. At every stage of a computation the finite control of T is in one of a finite set of states, the tape head scans one cell, and each cell contains one out of a finite set of symbols. Each step of a computation depends on the state (of the finite control) and on the symbol currently scanned. The following actions are taken at each step: M goes into a new state, a new symbol is printed by the tape head in the cell scanned, and the tape head is moved one position to the left or right.



Formally, we have

Definition 4.5: (V) A Turing machine is a quintuple $(Q, \Gamma, \delta, q_0, F)$, where

(i) Q is a finite set, the set of *(internal) states*;

(ii) q_0 is an element of Q, the *initial state*;

(iii) F is a subset of Q, the set of *accepting states*;

(iv) Γ is an alphabet, the set of *tape symbols*; one element of Γ is the *blank symbol* $\sqcup$; moreover, $Q \cap \Gamma = \emptyset$;

(v) δ, the *transition function*, is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$; it specifies the next step in a computation, depending on the current state and the tape symbol currently scanned.

We take the position that the word *blank (symbol)* has an absolute meaning, that is, there is one fixed symbol $\sqcup$ which serves as the blank symbol in every TM. Thus $\sqcup$ need not be specified anew whenever a TM is defined.

A *configuration* of a TM M is determined by the state of M, the contents of each tape cell and the position of the tape head. However, two configurations are considered identical, if they differ only by shifting the tape contents and the tape head by a fixed number of positions. Also, we only consider configurations where almost all cells contain the blank symbol (although there are some works about TM-computations where this restriction does not apply). Let q be the state of M, and c the cell scanned by the tape head. Then all essential information is contained in the triple (w_l, q, w_r), where w_l is the sequence of tape symbols from the leftmost non-blank up to the cell immediately to the left of c, and w_r is the sequence of tape symbols from c to the rightmost non-blank; w_l and/or w_r are empty, if all tape cells to the left of c or all tape cells in c and to the right of c contain blanks. The configuration (w_l, q, w_r) is denoted by the single string $w_l q w_r$, but in this notation we also allow to include leading and/or trailing blanks. Thus $\sqcup \ldots \sqcup w_l q w_r \sqcup \ldots \sqcup$ denotes exactly the same configuration as $w_l q w_r$. Due to the condition $Q \cap \Gamma = \emptyset$ in definition 4.5 (iv), the string $w_l q w_r$ does indeed uniquely determine the configuration denoted. This leads to the following formal

Definition 4.6: Let T be a TM, with notation as in definition 4.5.

(i) A *configuration* (*instantaneous decription*) of T is a triple (w_l, q, w_r), where w_l, $w_r \in \Gamma^*$ and $q \in Q$; if w_l (w_r) is non-empty, then the first (last) symbol of w_l (w_r) is not $\sqcup$. Such a configuration is denoted by any string of the form $\sqcup^r w_l q w_r \sqcup^s$.

(ii) In what follows let q, q' etc. stand for states of T, a, b etc. for a tape symbols, w_l, w_r' etc. for strings of tape symbols. Let $\mathcal{C}$, $\mathcal{C}'$ be configurations of T. Then $\mathcal{C}$

yields $\mathcal{C}'$ in one step (of a computation by T) ($\mathcal{C} \underset{T}{\longrightarrow} \mathcal{C}'$), iff one of the following conditions holds for some q, q', a, b c, w_l, w_r:

(a) $\mathcal{C} = w_l q a w_r$, $\delta(q,a) = (q', b, R)$, and $\mathcal{C}' = w_l b q' w_r$.

(b) $\mathcal{C} = w_l c q a w_r$, $\delta(q,a) = (q', b, L)$, and $\mathcal{C}' = w_l q' c b w_r$.

(c) $\mathcal{C} = w_l q$, $\delta(q, \sqcup) = (q', b, R)$, and $\mathcal{C}' = w_l b q'$.

(d) $\mathcal{C} = w_l c q$, $\delta(q, \sqcup) = (q', b, L)$, and $\mathcal{C}' = w_l q' c b$.

Rules (a) and (b) also apply in the case where w_l is the empty string, hence the case where $\mathcal{C}$ is the configuration $q a w_r$ need not be considered separately.

(iii) Analogously as for URM's, a configuration $\mathcal{C}$ is a *halting configuration* of T, iff there exists no configuration $\mathcal{C}'$ such that $\mathcal{C} \underset{T}{\longrightarrow} \mathcal{C}'$. This is the case iff $\mathcal{C} = w_l q a w_r$ with $\delta(q,a)\uparrow$ or $\mathcal{C} = w_l q$ with $\delta(q, \sqcup)\uparrow$.

(iv) The terms 'computation of T starting in configuration $\mathcal{C}$,' 'final configuration', 'termination' and 'step of a computation' are defined exactly as in definition 1.4.

Two types of tasks can be performed by a TM T: *computing functions*, which is more relevant for the purpose of the present chapter, and *accepting languages*, which plays an even more important rôle in other investigations. In both cases it is natural to regard T as acting on strings rather than on natural numbers. These strings are formed from an alphabet $\Sigma \subseteq \Gamma - \{\sqcup\}$. The computation for input $(w_1, \ldots, w_n)$ starts in the configuration $q_0 w_1 \sqcup w_2 \sqcup \ldots \sqcup w_n$. The blank symbol serves a special purpose and must not be an element of Σ. The elements of $\Gamma - \Sigma$ are auxiliary symbols used for scratchwork.

We also wish to associate with T a function $f\colon \omega^n \to \omega$ and a set $A \subseteq \omega^n$ computed (accepted) by T ($n \in \omega$). This is done in the obvious way, using the one-one correspondence between Σ^* and ω. Thus we proceed just as in section 2.3, except that the rôles of the data types ω and Σ^* are now swapped.

Notation: If $f\colon \omega^n \to \omega$ and $A \subseteq \omega^n$, then f_Σ and A_Σ denote the corresponding function from $(\Sigma^*)^n$ to Σ^* and the corresponding subset of $(\Sigma^*)^n$ respectively. Thus f_Σ and A_Σ are the uniquely determined function g and set B such that $f = g_\omega$ and $A = B_\omega$ respectively.

Definition 4.7: (V) Let T be a TM (with Q, q_0, Γ etc. as above).

(i) The computation of T for input $\vec{w}_n \in (\Sigma^*)^n$ ($\Sigma \subseteq \Gamma - \{\sqcup\}$) is the computation starting in the configuration $q_0 w_1 \sqcup w_2 \sqcup \ldots \sqcup w_n$.

(ii) T *computes* a partial function $f\colon (\Sigma^*)^n \to \Sigma^*$, iff the following holds for every input $\vec{w}_n \in (\Sigma^*)^n$: If $f(\vec{w}_n)\downarrow$, then the computation of T for input $\vec{w}_n$ terminates in a configuration where the sequence of tape symbols from the leftmost to the rightmost non-blank is $f(w)$; this means that $f(w)$ is the *result* of the computation. On the other hand, if $f(w)\uparrow$, then this computation does not terminate. A function $f\colon (\Sigma^*)^n \to \Sigma^*$ is *T-computable*, iff f is computed by some TM.

(iii) T *accepts* a set $A \subseteq (\Sigma^*)^n$, iff the following holds for every input $\vec{w}_n \in (\Sigma^*)^n$: If $\vec{w}_n \in A$, then the computation of T for input $\vec{w}_n$ terminates in a configuration where the state of T is an accepting state (element of F); but if $\vec{w}_n \notin A$, then this computation either does not terminate or terminates in a configuration where the state of T is not an accepting state. If in addition T halts for every input

$\vec{w}_n \in (\Sigma^*)^n$, then T *decides* A. A set $A \subseteq (\Sigma^*)^n$ is *T-acceptable* (*T-decidable*), iff it is accepted (decided) by some TM.

(iv) The computation of T for input $\vec{x}_n \in \omega^n$ with respect to an enumeration $(\sigma_0, \ldots, \sigma_{m-1})$ of Σ is the same as the computation for input $\big(\mathrm{cd}_{\Sigma^*}^{-1}(x_1), \ldots, \mathrm{cd}_{\Sigma^*}^{-1}(x_n)\big)$. (Recall that the mapping cd_{Σ^*} from definition 2.11 depends on $(\sigma_0, \ldots, \sigma_{m-1})$.) A function $f{:}\, \omega^n \to \omega$ is computed by T with respect to $(\sigma_0, \ldots, \sigma_{m-1})$ or T-computable with respect to $(\sigma_0, \ldots, \sigma_{m-1})$ respectively, iff f_Σ is computed by T (T-computable). A set $A \subseteq \omega^n$ is accepted by T (decided by T, T-acceptable, T-decidable) with respect to $(\sigma_0, \ldots, \sigma_{m-1})$, iff this property (without the specification 'with respect to Σ) is satisfied by A_Σ. The clause 'with respect to $(\sigma_0, \ldots, \sigma_{m-1})$' may be omitted if $(\sigma_0, \ldots, \sigma_{m-1})$ is given in the context.

When a TM $T = (Q, \Gamma, \delta, q_0, F)$ is defined, and also in general TM-constructions like the proof of theorem 4.4, the following conventions apply unless otherwise stated: The elements of Q are symbols like q, q', $\overline{q}$ etc., possibly with subscripts and/or superscripts, and q_0 is the initial state. Other lowercase roman letters, digits and special characters like \$, # etc. are tape symbols. We also use symbols like q, q' etc. to denote an *arbitrary* state, and symbols like a, b, a' etc. to denote *arbitrary* tape symbols (compare the remark after exercise 1.1). In the latter case Greek letters are also used occasionally, and the symbol D represents an *arbitrary direction* (element of $\{L, R\}$).

In general the only components of T that have to be specified are the *transitions*, that is, the values of δ which are defined, and the set F in case T is designed to accept a set. Then it is understood that $\delta(q, a)$ is undefined whenever this value is not given, that Q and Γ consist of those state- and tape symbols respectively which occur in transitions, and that $F = \emptyset$ if T is designed to compute a function.

Example 4.2: We wish to design a TM M computing $\lambda w.ww$. The construction is generic, so that it works for any input alphabet Σ. For every $a \in \Sigma$, $\tilde{a}$ is a new symbol $\notin \Sigma$. If $\Sigma = \{1\}$, then M also computes the integer function $\lambda x.2x$ with respect to (1). For input w, M works as follows: The first input symbol a is marked, that is, replaced by $\tilde{a}$, and $\tilde{a}$ is also attached at the right. Then the same is done with every other input symbol. This work is finished when no element of Σ is left. Finally, all $\tilde{a}$ are converted back to a. M ultimately halts in state q_3, except for the case $w = \epsilon$, when the computation terminates instantly. The following transitions are needed for all $a \in \Sigma$ (and $b \in \Sigma$).

$$
\begin{array}{rl}
\delta(q_0, a) = (q_a, \tilde{a}, R) & \text{mark a symbol} \\
\left.\begin{array}{r} \delta(q_a, b) = (q_a, b, R) \\ \delta(q_a, \tilde{b}) = (q_a, \tilde{b}, R) \end{array}\right\} & \text{move right} \\
\delta(q_a, \sqcup) = (q_1, \tilde{a}, L) & \text{attach } \tilde{a} \text{ at the right} \\
\delta(q_1, \tilde{a}) = (q_1, \tilde{a}, L) & \text{move left through attached symbols} \\
\left.\begin{array}{r} \delta(q_1, a) = (q_2, a, L) \\ \delta(q_2, a) = (q_2, a, L) \end{array}\right\} & \text{move left through unmarked symbols} \\
\delta(q_2, \tilde{a}) = (q_0, \tilde{a}, R) & \text{scanning leftmost unmarked symbol, start new cycle} \\
\delta(q_1, \sqcup) = (q_3, \sqcup, R) & \text{no unmarked symbol left} \\
\delta(q_3, \tilde{a}) = (q_3, a, R) & \text{`unmark' symbols, that is convert } \tilde{a} \text{ back to } a
\end{array}
$$

For input $aba \in \{a,b\}^*$ M goes through the following sequence of configurations:

$$\begin{array}{lll}
q_0aba & \tilde{a}\tilde{b}aq_1\tilde{a}\tilde{b} & \tilde{a}q_1\tilde{b}\tilde{a}\tilde{a}\tilde{b}\tilde{a} \\
\tilde{a}q_aba & \tilde{a}\tilde{b}q_1a\tilde{a}\tilde{b} & q_1\tilde{a}\tilde{b}\tilde{a}\tilde{a}\tilde{b}\tilde{a} \\
\tilde{a}bq_aa & \tilde{a}q_2\tilde{b}a\tilde{a}\tilde{b} & q_1\sqcup\tilde{a}\tilde{b}\tilde{a}\tilde{a}\tilde{b}\tilde{a} \\
\tilde{a}baq_a & \tilde{a}\tilde{b}q_0a\tilde{a}\tilde{b} & q_3\tilde{a}\tilde{b}\tilde{a}\tilde{a}\tilde{b}\tilde{a} \\
\tilde{a}bq_1a\tilde{a} & \tilde{a}\tilde{b}\tilde{a}q_a\tilde{a}\tilde{b} & aq_3\tilde{b}\tilde{a}\tilde{a}\tilde{b}\tilde{a} \\
\tilde{a}q_2ba\tilde{a} & \tilde{a}\tilde{b}\tilde{a}\tilde{a}q_a\tilde{b} & abq_3\tilde{a}\tilde{a}\tilde{b}\tilde{a} \\
q_2\tilde{a}ba\tilde{a} & \tilde{a}\tilde{b}\tilde{a}\tilde{a}\tilde{b}q_a & abaq_3\tilde{a}\tilde{b}\tilde{a} \\
\tilde{a}q_0ba\tilde{a} & \tilde{a}\tilde{b}\tilde{a}\tilde{a}q_1\tilde{b}\tilde{a} & abaaq_3\tilde{b}\tilde{a} \\
\tilde{a}\tilde{b}q_ba\tilde{a} & \tilde{a}\tilde{b}\tilde{a}q_1\tilde{a}\tilde{b}\tilde{a} & abaabq_3\tilde{a} \\
\tilde{a}\tilde{b}aq_b\tilde{a} & \tilde{a}\tilde{b}q_1\tilde{a}\tilde{a}\tilde{b}\tilde{a} & abaabaq_3 \\
\tilde{a}\tilde{b}a\tilde{a}q_b & &
\end{array}$$

Exercise 4.8: [4] Design a TM which accepts $\{\, x \in \omega \mid x \text{ is a perfect square} \,\}$ with respect to (1).

The way Turing machines are specified as computing functions $f \in \mathcal{NF}$ is yet another convention which differs considerably from textbook to textbook; the one from definition 4.7 (iv) comes close to that in Brainerd/Landweber [2]. In general it is more common to encode an input vector $(x_1, \ldots, x_n) \in \omega^n$ by the string $1^{x_1+1}\#1^{x_2+1}\#\ldots\#1^{x_n+1}$, or something similar, when TM's are used to compute integer functions. (Recall that by definition 2.12 (vii) 1^m is the string consisting of m 1's.) This scheme agrees roughly, but not completely, with the one from definition 4.7 (iv) for $\Sigma = \{1\}$.

If a TM is to be regarded as a machine, then certain additional data have to be specified, namely the input- and output alphabet Σ, the number of arguments n of the function computed, and whether this function is from $(\Sigma^*)^n$ to Σ^* or from ω^n to ω.

At this stage it should be clear that the design of TM's for any but the simplest tasks is quite tedious, since the instruction set is so primitive. In order to facilitate TM-constructions, we introduce certain *macros*, namely abbreviations for certain sets of transitions.

rsubst$(q, \Delta, \Theta, S, q')$: Here Δ and Θ are sets of tape symbols, and S is a set of pairs (b, b') of tape symbols. The first components b must all be different. Denote the set of first components of elements of S by S_1. The meaning of the macro is: if the TM is in state q, scanning symbol $a \in \Delta$, then move right to the next occurrence of a tape symbol $a' \in \Theta$, replace every symbol $b \in S_1$ on the way by the corresponding b', and go into state q'. This is effected by the following set of transitions:

$$\begin{array}{rl}
\delta(q, a) = (q', a, R) & \text{for all } a \in \Delta - S_1 \\
\delta(q, a) = (q', a', R) & \text{for every pair } (a, a') \in S \text{ such that } a \in \Delta \\
\delta(q', b) = (q', b, R) & \text{for all } b \notin \Theta \cup S_1 \\
\delta(q', b) = (q', b', R) & \text{for all pairs } (b, b') \in S \text{ such that } b \notin \Theta.
\end{array}$$

This works under the assumption that T leaves state q' right after performing $rsubst(q, \Delta, \Theta, S, q')$ and does not enter q' otherwise; this quarantees that there is no conflict with the given transitions $\delta(q', b) = \ldots$. This condition is satisfied for the applications of *rsubst* below, and similarly for some of the other macros to be introduced.

If Δ or Θ is a one-element set $\{a\}$, then a itself instead of Δ or Θ may be used as parameter; this conventions also applies to the other macros defined below. When a macro is applied in the construction of a TM, then the formal parameters have to be replaced by actual ones. Such a macro call expands into a sequence of transitions in the following way: Use the definition of the macro, but replace every symbol which represents a state but is not a parameter (like q_i' in the definition of *rmove.r*) by a state symbol which is not used anywhere else in the construction of the TM.

$rmove(q, \Delta, \Theta, q')$: If M is in state q scanning a symbol in Δ, then move right to the next occurrence of a symbol in Θ and go into state q'. This is the same as $rsubst(q, \Delta, \Theta, \emptyset, q')$.

$rshift(q, \Delta, a, \Theta, q', D)$: If M is in state q scanning a tape symbol in Δ, then shift every symbol from the current position of the reading/writing head (position i) up to the next occurrence of a symbol in Θ (position j) one cell to the right. The symbol a goes into position i, whereas the symbol originally in position j disappears. At the end the reading/writing head is placed one position to the right or to the left of position j, depending on D:

$$\begin{aligned}
\delta(q, b) &= (q_b, a, R) && \text{for all } b \in \Delta \\
\delta(q_b, c) &= (q_c, b, R) && \text{for all } b, c \notin \Theta) \\
\delta(q_b, c) &= (q', b, D) && \text{for all } b \notin \Theta \text{ and } c \in \Theta
\end{aligned}$$

$rsubst.r(q, \Delta, \Theta, S, q')$, $rmove.r(q, \Delta, \Theta, q')$, $rshift.r(q, \Delta, a, \Theta, q', D)$: These are like $rsubst(q, \Delta, \Theta, S, q')$, $rmove(q, \Delta, \Theta, q')$, $rshift(q, \Delta, a, \Theta, q', D)$, except that M has to move up to the r-th occurrence of a symbol in Θ to the right of the current position, including the current position itself. They are not very important for TM's in general, but they play an important rôle in the proof of theorem 4.4. $rsubst.r(q, \Delta, \Theta, S, q')$ is defined by the following transitions:

$$\begin{aligned}
& rsubst(q, \Delta - \Theta, \Theta, S, q_1') \\
& rsubst(q, \Delta \cap \Theta, \Theta, S, q_2') \\
& rsubst(q_1', \Theta, \Theta, S, q_2') \\
& \quad \vdots \\
& rsubst(q_{r-2}', \Theta, \Theta, S, q_{r-1}') \\
& rsubst(q_{r-1}', \Theta, \Theta, S, q')
\end{aligned}$$

This definition works for $r \geq 2$. In the case $r = 0$ the TM just has to go from state q into q'. This requires no transitions for $q = q'$, and for $q \neq q'$ the corresponding transitions are

$$\begin{aligned}
\delta(q, a) &= (q'', a, R) && \text{for } a \in \Delta \\
\delta(q'', a) &= (q', a, L) && \text{for all } a \in \Gamma.
\end{aligned}$$

The case $r = 1$ is also straightforward:

$$\begin{array}{ll} rsubst(q, \Delta - \Theta, \Theta, S, q') & \\ \delta(q, a) = (q'', a, R) & \text{for } a \in \Delta \cap \Theta \\ \delta(q'', a) = (q', a, L) & \text{for all } a \in \Gamma. \end{array}$$

We note that *rsubst*.1 is not quite the same as *rsubst*, since $rsubst.1(q, \Delta, \Theta, S, q')$ leaves the tape head in the original position, if it contains a symbol belonging to Θ.

$rmove.r(q, \Delta, \Theta, q')$ is again equivalent to $rsubst.r(q, \Delta, \Theta, \emptyset, q')$. We omit the definition of $rshift.r(q, \Delta, a, \Theta, q', D)$ for $r = 0$; for $r \geq 1$ it is given by the following transitions:

$$\begin{array}{rl} \delta(q, b) = (q_{b,0}, a, R) & \text{for all } b \in \Delta - \Theta \\ \delta(q, b) = (q_{b,1}, a, R) & \text{for all } b \in \Delta \cap \Theta, \text{ if } r \geq 2 \\ \delta(q, b) = (q', a, D) & \text{for all } b \in \Delta \cap \Theta, \text{ if } r = 1 \\ \delta(q_{b,i}, c) = (q_{c,i}, b, R) & \text{for all } b \in \Gamma,\ c \notin \Theta \text{ and } 0 \leq i < r \\ \delta(q_{b,i}, c) = (q_{c,i+1}, b, R) & \text{for all } b \in \Gamma,\ c \in \Theta \text{ and } 0 \leq i < r - 1 \\ \delta(q_{b,r-1}, c) = (q', b, D) & \text{for all } b \in \Gamma \text{ and } c \in \Theta, \text{ if } r \geq 2 \end{array}$$

lsubst, *lmove*, *lsubst.r*, *lmove.r*, *lshift*, *lshift.r*: These macros are like *rsubst* etc., except that the movement of the tape head and the shift of symbols are to the left. It is obvious how they have to be defined.

Exercise 4.9: [2] Redo example 4.2 using the macros introduced.

Exercise 4.10: [4] Definition 4.7 (ii) may be modified by declaring that the result of a computation is x iff the following holds: If in the final configuration the sequence of tape symbols from the leftmost to the rightmost non-blank is x', then x is the longest prefix of x' which is an element of Σ^*. In other words, the TM is allowed to leave some 'garbage' on the tape to the right of the output. This modified convention is perhaps somewhat less natural than the original one, but it has the advantage that *every* TM computes a partial function from $(\Sigma^*)^n$ to Σ^* for every n and every $\Sigma \subseteq \Gamma$.

Prove that with this modification the set of T-computable functions is not changed.

The following theorem provides the next step towards proving the equivalence of various models of computation.

Theorem 4.4: Let $(\sigma_0, \ldots, \sigma_{m-1})$ be an enumeration of an alphabet Σ. Then every partial recursive function $f\colon \omega^n \to \omega$ is T-computable with respect to $(\sigma_0, \ldots, \sigma_{m-1})$. (Hence the same implication holds of course for the corresponding function f_Σ, see definition 4.7 (iv)). (The converse of this statement will be contained in theorem 4.5.)

Proof: Let $f\colon \omega^n \to \omega$ be a given partial recursive function. By theorem 2.2 f is computed by some $\mathcal{GL}$-program P with respect to variables $X_1, \ldots, X_n, X_{n+1}$. We may assume that the output variable X_{n+1} does not occur among the input variables, for if we are given a program which computes f with respect to $X_1, \ldots, X_n$ and X_i $(1 \leq i \leq n)$, then we may choose a new variable X_{n+1}, attach the statement $X_{n+1} := X_i$, and eliminate this assignment by using the method of theorem 2.1. (This elimination introduces a **while**-statement, which is subsequently replaced by **goto**.) Henceforth P

denotes the program modified in this fashion. We may assume that the set of variables of P is $R = \{X_1, \ldots, X_k\}$ $(k \geq n+1)$. Also, let $I_0, \ldots, I_{l-1}$ be the instructions of P.

With P we associate a URM $U = (R, P)$, as it was done for $\mathcal{WL}$-programs in definition 1.2 (vii) for $\mathcal{WL}$-programs. U, as well as any TM, may be regarded as machines, see section 4.2. We construct a TM T which simulates U. The configuration $\mathcal{C} = (j, V)$ of U with $V(X_i) = x_i$ corresponds to the following configuration $\mathcal{C}'$ of T, where we write w_i for $\mathrm{cd}_{\Sigma^*}^{-1}(x_i)$:

$$q_j w_1 \sqcup w_2 \sqcup \ldots \sqcup w_k.$$

Thus we have $h(\mathcal{C}') = \mathcal{C}$, where h is the mapping from definition 4.4. According to our conventions, the computation of T for input $\vec{x}_n$ already starts in the configuration $q w_1 \sqcup w_2 \sqcup \ldots \sqcup w_n$, which corresponds to the initial configuration of U for input $\vec{x}_n$. (Since all X_i with $i > n$ are initialized to 0, the correponding strings w_i are empty.) Now T needs transitions for simulating each instruction I_t $(t < l)$, starting in state q_t.

(a) Assume that I_t is of the form $X_r := X_r + 1$. In order to simulate it, T has to work as follows:

(i) Move the tape head to the $\sqcup$ to the right of w_r.
(ii) Search leftwards for a symbol σ_i of w_r such that $i \leq m-2$; if no such symbol is found, then stop at the $\sqcup$ to the left of w_r.
(iii) If a symbol σ_i with $i \leq m-2$ is found at step (ii), then replace it by σ_{i+1}, replace every occurrence of σ_{m-1} in w_r to the right of it by σ_0 and go to (v).
(iv) If no such σ_i is found in (ii), then replace every every occurrence of σ_{m-1} in w_r by σ_0, add another copy of σ_0, and shift the portion of the tape to the right of w_r one position to the right.
(v) Finally, move the tape head back to the leftmost non-blank.

The following transitions are needed to perform this task:

$$\begin{array}{ll}
\mathit{rmove}.r(q_t, \Sigma \cup \{\sqcup\}, \sqcup, q_{t,0}) & \\
\mathit{lmove}(q_{t,0}, \sqcup, \{\sigma_0, \ldots, \sigma_{m-2}, \sqcup\}, q_{t,1}) & \\
\delta(q_{t,1}, \sigma_i) = (q_{t,2}, \sigma_{i+1}, R) & \text{for } 0 \leq i \leq m-2 \\
\mathit{rsubst}(q_{t,2}, \sigma_{m-1}, \sqcup, \{(\sigma_{m-1}, \sigma_0)\}, q_{t,3}) & \\
\mathit{lmove}.(r+1)(q_{t,3}, \sqcup, \sqcup, q_{t,6}) & \\
\mathit{rsubst}(q_{t,1}, \sqcup, \sqcup, \{(\sigma_{m-1}, \sigma_0)\}, q_{t,4}) & \\
\mathit{rshift}.(k-r+1)(q_{t,4}, \sqcup, \sigma_0, \sqcup, q_{t,5}, R) & \\
\mathit{lmove}.(k+1)(q_{t,5}, \sqcup, \sqcup, q_{t,6}) & \\
\delta(q_{t,6}, \sqcup) = (q_{t+1}, \sqcup, R) &
\end{array}$$

(b) If I_t is of the form $X_r := X_r \dot{-} 1$, then the simulation proceeds similarly as in (a). The changes are as follows:

(i) The rôles of σ_0 and σ_{m-1} are swapped.
(ii) The tape is not changed, if $w_r = \epsilon$.
(iii) If w_r consists entirely of σ_0's, then one of these has to be deleted, the rest have to be changed to σ_{m-1}, and the portion of the tape to the right of w_r has to be shifted one position to the left.

The details are left for exercise 4.11.

(c) If I_t is the instruction **if** $X_r > 0$ **then goto** L, and I_i is the instruction labelled L, then the simulation of I_t is easy:

$$\begin{aligned}
&rmove.(r-1)(q_t, \Sigma \cup \{\sqcup\}, \sqcup, q_{t,0}) \\
&\delta(q_{t,0}, \sqcup) = (q_{t,1}, \sqcup, R) \\
&lmove.r(q_{t,1}, \Sigma, \sqcup, q_{t,2}) \\
&lmove.(r+1)(q_{t,1}, \sqcup, \sqcup, q_{t,3}) \\
&\delta(q_{t,2}, \sqcup) = (q_i, \sqcup, R) \\
&\delta(q_{t,3}, \sqcup) = (q_{t+1}, \sqcup, R)
\end{aligned}$$

This works for $r > 1$; in the case $r = 1$ the first four lines have to be replaced by

$$\begin{aligned}
&\delta(q_t, a) = (q_{t,2}, a, L) \quad \text{for } a \in \Sigma \\
&\delta(q_t, \sqcup) = (q_{t,3}, \sqcup, L)
\end{aligned}$$

The simulation finally terminates if and when state q_l is reached, which corresponds to a configuration of U where the instruction counter has value l. Now the result of the computation of U is the value of X_{n+1}. In order that the computation of T gives the same result, everything except w_{n+1} has to be erased:

$$\begin{aligned}
&rsubst.n(q_l, \Sigma \cup \{\sqcup\}, \sqcup, S, q_{l,0}) \quad \text{where } S = \{\, (a, \sqcup) \mid a \in \Sigma \,\} \\
&rmove(q_{l,0}, \sqcup, \sqcup, q_{l,1}) \\
&rsubst.(k-n)(q_{l,1}, \sqcup, \sqcup, S, q_f) \quad \text{with the same } S \text{ as above}
\end{aligned}$$

The computation of T terminates in state q_f. This completes definition of T and the proof. □

Exercise 4.11: [3] List the transitions necessary to simulate an instruction I_t of U of the form $X_r := X_r \dot{-} 1$.

4.4. Markov algorithms.

We now come to the last of the models of computation to be considered in detail in these lecture notes. Like TM's, *Markov algorithms* (MA's) process words rather than numbers. Here the stepwise computation is defined in an even simpler fashion than for URM's and TM's, since the configuration at every stage is just a string, and each step consists simply in replacing a substring of the current string by another, according to certain rules which define the MA. MA's are closely related to grammars, Thue systems and other rewriting systems, some of which will be dealt with in (DCC).

Definition 4.8: (i) A *Markov algorithm* (MA) M over an alphabet Γ containing $\sqcup$ is a finite sequence of *instructions* of the form

$$\begin{gathered}
u_0 \to v_0 \\
\vdots \\
u_{n-1} \to v_{n-1}
\end{gathered}$$

with u_i, $v_i \in \Gamma^*$.

(ii) A *configuration* of M is an element of Γ^*.

(iii) Let w, w' be configurations of M, where M is as in (i). w *yields* w' in one step ($w \underset{M}{\longrightarrow} w'$), iff there exist an index i and strings x, y such that $w = xu_iy$, $w' = xv_iy$, i is the smallest index such that u_i is a substring of w, and x is the shortest prefix of w such that w is of the form xu_iy. (More informally, w' is obtained from w by choosing the smallest i such that u_i is a substring of w and replacing the leftmost occurrence of u_i by v_i.)

(iv) The terms 'halting configuration', 'computation of M starting in configuration w,' 'final configuration', 'termination' and 'step of a computation' are again defined as in definitions 1.4 and 4.6. In the notation from (i), w is a halting configuration of M iff no u_i is a substring of w.

(v) The computation of M for input $\vec{w}_n \in (\Sigma^*)^n$ ($\Sigma \subseteq \Gamma - \{\sqcup\}$) is the computation starting in the configuration $\sqcup w_1 \sqcup w_2 \sqcup \ldots \sqcup w_n$.

(vi) M *computes* a partial function $f\colon (\Sigma^*)^n \to \Sigma^*$, iff the following holds for every input $\vec{w}_n \in (\Sigma^*)^n$: If $f(\vec{w}_n)\downarrow$, then the computation of M for input $\vec{w}_n$ terminates in the configuration $f(\vec{w}_n)$; this means that $f(\vec{w}_n)$ is the *result* of the computation. On the other hand, if $f(\vec{w}_n)\uparrow$, then this computation does not terminate. A function $f\colon (\Sigma^*)^n \to \Sigma^*$ is *M-computable*, iff f is computed by some MA.

(vii) M *accepts* $A \subseteq (\Sigma^*)^n$, iff A is the set of all $\vec{w}_n \in (\Sigma^*)^n$ such that the computation of M for input $\vec{w}_n$ terminates (that is, A is the domain of the function computed by M).

(viii) The computation of M for input $\vec{x}_n \in \omega^n$ with respect to $(\sigma_0, \ldots, \sigma_{m-1})$ is the same as the one for input $\big(\mathrm{cd}_{\Sigma^*}^{-1}(x_1), \ldots, \mathrm{cd}_{\Sigma^*}^{-1}(x_n)\big)$. A function $f\colon \omega^n \to \omega$ is computed by M with respect to $(\sigma_0, \ldots, \sigma_{m-1})$ or M-computable with respect to $(\sigma_0, \ldots, \sigma_{m-1})$ respectively, if f_Σ is computed by M (M-computable). A set $A \subseteq \omega^n$ is accepted by M (M-acceptable) with respect to $(\sigma_0, \ldots, \sigma_{m-1})$, iff this property (without the specification 'with respect to Σ') is satisfied by A_Σ. The specification 'with respect to $(\sigma_0, \ldots, \sigma_{m-1})$' may be omitted if $(\sigma_0, \ldots, \sigma_{m-1})$ is given in the context.

Remark 4.1: A few comments concerning this definition are in order. If M contains an instruction of the form $(\epsilon \to v_i)$, then no computation by M can terminate, since every string contains ϵ as a subword. The symbol $\sqcup$ plays a slightly different rôle than for TM's: Now there is no such thing as the rest of the tape filled with blanks, and $\sqcup \ldots \sqcup w \sqcup \ldots \sqcup$ is no longer the same configuration as w. A leading $\sqcup$ is included in the initial configuration, in addition to the blanks which separate the input words. This is necessary only for $n = 0$ and for $n = 1$ with input ϵ. In these two cases the computation would otherwise start in the configuration ϵ, and it would either be non-terminating (if $u_i = \epsilon$ for some i), or yield the result ϵ after 0 steps (if $u_i \neq \epsilon$ for all i). Definition 4.8 is set up in such a way that $\sqcup$ is the only special symbol that has to be disallowed in input strings. The first part of the proof of theorem 4.5 below (every TM is simulated by some MA) would be slightly simpler if the leading $\sqcup$ were replaced by some other special symbol like \$; in this case blanks could simply be eliminated by $\sqcup \to \epsilon$ at the end of the computation.

Definition 4.8 (vii) differs from definition 4.7 (iii), since an MA M has no accepting or non-accepting states. For the same reason there is no natural way to define M as *deciding* a set. The set accepted by a URM could be defined analogously, see the remark before definition 1.6.

Like TM's, MA's can be regarded as machines if certain additional specifications are given, see the corresponding remark about TM's in the previous section.

In examples of MA's the alphabet Γ will usually not be specified; unless otherwise stated, Γ is the set of all characters occurring in instructions.

Example 4.3: We design an MA M which computs $\lambda w.ww^R$, where w^R is the string w read from right to left. Similarly as in example 4.2 we give a definition which works for every input alphabet Σ, assuming that $\tilde{\alpha} \notin \Sigma$ for $\alpha \in \Sigma$, and M also computes the integer function $\lambda x.2x$ with respect to (1). M works as follows: The blank is swapped with the input symbol α to the right of it, and a new marked symbol $\tilde{\alpha}$ is introduced; $\tilde{\alpha}$ is moved to the right past the unmarked symbols. This cycle is repeated for all input symbols. Thereafter the blank is deleted, and all symbols are unmarked. This is illustrated by the following sample computation:

$$\begin{array}{ll}
\sqcup abb & abb\sqcup\tilde{b}\tilde{b}\tilde{a} \\
a\sqcup\tilde{a}bb & abb\tilde{b}\tilde{b}\tilde{a} \\
a\sqcup b\tilde{a}b & abbb\tilde{b}\tilde{a} \\
a\sqcup bb\tilde{a} & abbbb\tilde{a} \\
ab\sqcup\tilde{b}b\tilde{a} & abbbba \\
ab\sqcup b\tilde{b}\tilde{a} &
\end{array}$$

This is accomplished by the following instructions:

(i) $\tilde{\alpha}\beta \to \beta\tilde{\alpha}$ for all $\alpha,\ \beta \in \Sigma$,
(ii) $\sqcup\alpha \to \alpha\sqcup\tilde{\alpha}$ for all $\alpha \in \Sigma$,
(iii) $\sqcup \to \epsilon$,
(iv) $\alpha\tilde{\beta} \to \alpha\beta$ for all $\alpha,\ \beta \in \Sigma$.

Of course the order in which these instructions are listed plays a crucial rôle: (ii) must be applied only when (i) cannot be applied, and the elimination of $\sqcup$ and the unmarking of marked symbols should be carried out only when neither (i) nor (ii) applies. True, within each of the groups (i)–(iv) the order is irrelevant.

Exercise 4.12: [3] Why are the instructions in the last group from example 4.3 of the form $\alpha\tilde{\beta} \to \alpha\beta$ rather than simply $\tilde{\beta} \to \beta$?

Exercise 4.13: [2] Design an MA which for input $w \in \{a, b, c\}^*$ computes

$$f(w) = \begin{cases} a & \text{if } w \text{ contains more } a\text{'s than } b\text{'s,} \\ b & \text{if } w \text{ contains at least as many } b\text{'s as } a\text{'s.} \end{cases}$$

We are now in a position to complete the proof of the equivalence of the models of computation. In the following theorem we also summarize previous results of this chapter.

Theorem 4.5: Let f be a partial function from ω^n to ω, $(\sigma_0, \ldots, \sigma_{m-1})$ an enumeration of an alphabet Σ. The following assertions are equivalent:

(i) f is partial recursive.

(ii) f can be defined from initial functions by (repeated application of) composition, primitive recursion and minimization.

(iii) f is T-computable with respect to $(\sigma_0, \ldots, \sigma_{m-1})$.

(iv) f is M-computable with respect to $(\sigma_0, \ldots, \sigma_{m-1})$.

(Of course, the theorem may equivalently be stated for functions $f\colon (\Sigma^*)^n \to \Sigma^*$, with the specification 'with respect to $(\sigma_0, \ldots, \sigma_{m-1})$' eliminated in (iii) and (iv). Note also that the result of exercise 2.17 is a corollary of this theorem.)

Proof: In view of theorems 4.1 and 4.4 it only remains to show (iii) $\Longrightarrow$ (iv) and (iv) $\Longrightarrow$ (i). We prove these implications for every function $f\colon (\Sigma^*)^n \to \Sigma^*$.

(iii) $\Longrightarrow$ (iv): Given a TM $T = (Q, \Gamma, \delta, q_0, F)$ which computes f, we construct a Markov algorithm M which simulates T. The function h from definiton 4.4 is defined as follows: If a configuration $\mathcal{C}$ of T is represented by the string $w = w_l q w_r$ (as explained in definition 4.6 (i)), and $w_r \neq \epsilon$, then $h(w) = \mathcal{C}$. (We could also write $h(w) = w$, but then it must be understood that at the left side the string w is itself a configuration of M, whereas at the right side it only *represents* a configuration of T.) The computation Ω' of M for input $\vec{w}_n$ starts in the configuration $\sqcup w_1 \sqcup \ldots \sqcup w_n$ and begins by replacing the initial $\sqcup$ by q_0, so that the (string representing the) initial configuration of T for that input is obtained. Then every step of the computation Ω of T is simulated by one single step of Ω'. For instance, if the configuration $w_l q a w_r$ occurs in Ω and $\delta(q, a) = (q', b, R)$, then the next step of the computation leads to the configuration $w_l b q' w_r$, which is obtained by replacing qa by bq' in the corresponding configuration of Ω'. Thus for all instances of δ corresponding instructions of M have to be introduced. Some care is needed for configurations of T of the form $w_l q w_r$ with w_l or w_r empty. In fact, the corresponding configuration of M may begin with, but must not end with q; a blank must be attached if necessary. At the end of the computation M deletes extra symbols. The instructions of M are as follows:

(a) For every transition $\delta(q, a) = (q', b, R)$ the instructions $qac \to bq'c$ $(c \in \Gamma)$, followed by $qa \to bq' \sqcup$.

(b) For every transition $\delta(q, a) = (q', b, L)$ the instructions $cqa \to q'cb$ $(c \in \Gamma)$, followed by $qa \to q' \sqcup b$.

(c) For every state q and tape symbol a such that $\delta(q, a)$ is not defined the instruction $qa \to \#a$, where $\#$ is a new symbol $\notin Q \cup \Gamma$.

(d) The instructions $\#a \to a\#$ $(a \in \Sigma)$ and $\#\sqcup \to \#$, followed by $\# \to @$; $@$ is another new symbol $\notin Q \cup \Gamma$.

(e) The instructions $a@ \to @a$ $(a \in \Sigma)$ and $\sqcup @ \to @$, followed by $@ \to \epsilon$.

(f) The instructions $\sqcup a \to q_0 a$ $(a \in \Gamma)$, followed by $\sqcup \to q_0 \sqcup$.

In the first step an instruction from the group (f) is applied, since no other instruction applies; the initial configuration of M contains no $q \in Q$ and none of the symbols $\#$, $@$. Then T is simulated step by step, using instructions (a), (b). Note that an extra blank is added only at the end of a configuration, since in group (a) $qa \to bq' \sqcup$ occurs after the instructions $qac \to bq'c$, and similarly for the instructions in groups (b) and (f). When the simulation is finished, the instructions from groups (c) and (d) are applied,

$q \in Q$ is replaced by #, # is moved to the right with blanks being eliminated, and # is replaced by @. Finally, @ is moved left, the remaining blanks (to the left of the original position of #) are eliminated, and so is @. Now no instructions of M can be applied, so the computation is finished, and the result agrees with the sequence of non-blank symbols on the tape in the final configuration of T. Thus M does indeed compute the same partial function f as T.

(iv) $\Longrightarrow$ (i): Let $f\colon (\Sigma^*)^n \to \Sigma^*$ be computed by a Markov algorithm M over Γ. Then Γ contains $\Sigma \cup \{\sqcup\}$, so let $(\sigma_0, \ldots, \sigma_{k-1})$ be an enumeration of Γ with $k > m$ and $\sigma_m = \sqcup$. Let $u_0 \to v_0, \ldots, u_{l-1} \to v_{l-1}$ be the intructions of M. In order to prove that f is partial recursive, it suffices to show that M can be viewed as a machine computing f in the sense of definition 4.3. We denote this machine by M', since it is not the same mathematical object as the Markov algorithm M. The construction makes use of several parts of theorem 2.12; it works as follows:

Clearly, $I_{M'} = (\Sigma^*)^n$, $O_{M'} = \Sigma^*$ and $\mathcal{CF}_{M'} = \Gamma^*$. The function $\text{init}_{M'}\colon (\Sigma^*)^n \to \Gamma^*$ is defined by $\text{init}_{M'}(\vec{w}_n) = \sqcup w_1 \sqcup w_2 \ldots \sqcup w_n$. To show that this function is primitive recursive, consider the function $g\colon \omega \times \Sigma^* \to \omega$ defined by

$$g(x, w) = \begin{cases} m & \text{for } x = 0, \\ \pi^{\Sigma}(w, x \dot{-} 1) & \text{for } x > 0. \end{cases}$$

Then we have $\breve{g}\big(L^{\Sigma}(w) + 1, w\big) = \sqcup w \in \Gamma^*$ in the notation of theorem 2.11 (iii), with Γ in the rôle of Σ. Denote the left side of this equality by $g_1(w)$. Then $\text{init}_{M'}(\vec{x}_n)$ is obtained from the strings $g_1(w_i)$ $(i = 1, \ldots, n)$ by repeated concatenation.

Next define functions h_1, $h_2\colon \omega \to \Gamma^*$ and $h\colon \Gamma^* \to \omega$ as follows:

$$\left.\begin{aligned} h_1(x) &= u_x, \\ h_2(x) &= v_x \end{aligned}\right\} \qquad \text{for } x < l$$
$$h_1(x) = h_2(x) = \epsilon \qquad \text{for } x \geq l,$$
$$h(w) = (\mu z < l)[\text{substr}^{\Gamma}(h_1(z), w)].$$

These functions are primitive recursive; for h_1 and h_2 this follows from exercise 2.6, since $h_j(x) = \epsilon$ for almost all x. For a configuration w of M, $h(w)$ indicates which instruction has to be applied next. Now we get

$$\text{halt}_{M'}(w) \equiv h(w) = l,$$
$$\text{next}_{M'}(w) = \text{subst}^{\Gamma}\big(h_1 h(w), h_2 h(w), w\big).$$

Finally, $\text{res}_{M'}$ transforms a string $w \in \Gamma^*$, whose symbols happen to be elements of Σ, into a string $w' \in \Sigma^*$ consisting of the same symbols. This function can be represented similarly as $\text{init}_{M'}$: $\text{res}_{M'}(w) = \breve{g}_2\big(L^{\Gamma}(w), w\big)$, where

$$g_2(x, w) = \begin{cases} \pi^{\Gamma}(w, x) & \text{if } \pi^{\Gamma}(w, x) < m, \\ 0 & \text{if } \pi^{\Gamma}(w, x) \geq m. \end{cases}$$

(This time the alphabet Σ from theorem 2.12 (iii) coincides with the Σ from the current context.) This completes the construction of M' and the proof. □

Exercise 4.14: [4] It was noted after exercise 4.8 that the computation of integer functions by TM's is usually defined differently in textbooks than it was in these lecture notes. For instance, in Rogers [21] the definition is as follows: The computation of a TM M for input $\vec{x}_n$ is the computation starting in configuration $q_0 1^{x_1+1}\#1^{x_2+1}\#\ldots\#1^{x_n+1}$. M computes a function $f{:}\,\omega^n \to \omega$ such that $f(\vec{x}_n)$ is the total number of 1's on the tape in the final configuration of the computation for input $\vec{x}_n$; if this computation does not terminate, then $f(\vec{x}_n)\uparrow$.

Show that this definition leads to the same set of T-computable functions as definition 4.7.

As noted, TM's can also perform another task besides the computation of functions, namely deciding or accepting sets. The next theorem shows how these capabilities are related to notions of recursive function theory:

Theorem 4.6: Let A be a subset of ω^n or $(\Sigma^*)^n$, $(\sigma_0, \ldots, \sigma_{m-1})$ an enumeration of Σ. Then

(i) A is T-decidable (with respect to Σ, if $A \subseteq \omega^n$) if and only if A is recursive.

(ii) A is T-acceptable (with respect to Σ) if and only if A is r.e.

Proof: It suffices to consider the case $A \subseteq \omega^n$.

(i): Suppose A is recursive. Define $U = (R, P)$, $X_1, \ldots, X_k$ and l as in the proof of theorem 4.4, with c_A in place of f, and let T be the TM constructed from U in that proof. A TM T' which decides A is constructed similarly as T, with the following changes: T' contains two extra states, an accepting state q_a and a non-accepting state q_b. The transitions of T needed to reach the configuration corresponding to the initial configuration of U, and those needed to simulate the computation steps of U, are also transitions of T'. But when the configuration corresponding to the final configuration of U is reached, then T' behaves differently from T: T' halts in state q_a, if the result of the computation by T is $\neq \epsilon$, in state q_b otherwise. Hence the following transitions replace the last group of transitions in the proof of theorem 4.4:

$$\begin{aligned}
&rmove.n(q_l, \Gamma, \sqcup, q_{l,1})\\
&\delta(q_{l,1}, \sqcup) = (q_{l,2}, \sqcup, R)\\
&\delta(q_{l,2}, a) = (q_a, a, R) \quad \text{for } a \in \Sigma\\
&\delta(q_{l,2}, \sqcup) = (q_b, \sqcup, R)
\end{aligned}$$

Conversely assume that $T = (Q, \Gamma, \delta, q_0, F)$ is a TM which decides A with respect to $(\sigma_0, \ldots, \sigma_{m-1})$. Let M be the MA constructed in the proof of theorem 4.5, which simulates T. We construct an MA M' which computes c_A. M' uses two new symbols \$ and ¢. For input $\vec{x}_n$ M', like M, simulates the computation by T; hence M' also contains the instructions of groups (a), (b) and (f) from the construction of M. Then, if T halts in an accepting state, M' replaces that state by \$, eliminates all other symbols and finally replaces \$ by σ_0; thus the result of the computation is σ_0, corresponding to the number 1. If, on the other hand, M halts in a non-accepting state, then M' proceeds similarly, except that ¢ takes the place of \$, and this symbol is just eliminated at the end. So in this case the result is ϵ, corresponding to 0. This is accomplished by replacing the intructions (c)–(e) of M by the following new groups of instructions:

(c) The instructions $qa \to \$$ if $q \in F$ or $qa \to$ ¢ if $q \notin F$, for all q and a such that $\delta(q, a)$ is not defined.

(d) The instructions $\$a \to \$$ and $a\$ \to \$$ $(a \in \Gamma)$, followed by $a \to \sigma_0$.
(e) The instructions $\cent a \to \cent$ and $a\cent \to \cent$ $(a \in \Gamma)$, followed by $\cent \to \epsilon$.

(ii): Let A be r.e. Then there exists a partial recursive function f with domain A. Design a TM M which computes f with respect to $(\sigma_0, \ldots, \sigma_{m-1})$, as in the proof of theorem 4.4, and make every state of M an accepting state. If $\vec{x}_n \in A$, then M halts for input $\vec{x}_n$ and accepts this input, whereas if $\vec{x}_n \notin A$, then M does not halt for input $\vec{x}_n$. Thus M accepts A.

Conversely, assume that there is a TM $T = (Q, \Gamma, \delta, q_0, F)$ which accepts A. From T construct an MA M''. M'' is similar to the MA M' constructed in (i), but the sybol $\cent$ is not used, and the instructions involving $\cent$ are replaced by $qa \to qa$ for all $q \notin F$ and a such that $\delta(q, a)$ is undefined. If for some input $\vec{x}_n$ T halts in an accepting state, then M'' performs like the MA M' from (i) and yields the result σ_0. If T does not halt, then neither does M''. If T halts in a non-accepting state q, scanning symbol a, then M'' simulates T and then loops by the instruction $qa \to qa$. Thus M'' computes a partial function whose domain is A, the set accepted by T. Hence A is r.e. □

4.5. Et cetera.

As noted at the beginning of this chapter, there exist numerous other formalisms for introducing the notion of partial recursive function, which were not yet mentioned. Among them are minor variants of each of the formalisms considered so far. For instance, we may consider various classes $\mathcal{U}$ of URM's, with different sets of instructions being allowed. Theorems 2.1 and 2.2 show that in certain cases $\mathcal{U}$ is equivalent to the class of URM's from definition 1.2 (vii) in the sense that the functions computed by some $U \in \mathcal{U}$ are exactly the partial recursive functions.

The formalism of μ-recursion may be modified by using different sets of operations: Instead of composition one may apply explicit definition, as in theorem 2.4, or various forms of substitution. The latter kind of operation transforms $f(\vec{x}_n)$ into functions like the following:

$$\lambda x_1 \ldots x_{i-1} x_{i+1} \ldots x_n . f(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n),$$
$$\lambda x_1 \ldots x_{i-1} x_{i+1} \ldots x_n \vec{y}_m . f\big(x_1, \ldots, x_{i-1}, g(\vec{y}_m), x_{i+1}, \ldots, x_n\big),$$
$$\lambda \vec{x}_n . f(x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n)$$

etc. Also, primitive recursion may be replaced by pure iteration, see example 2.4 (vi). In some cases the set of initial functions has to be changed as well.

If f is defined from functions g, h, … by composition or primitive recursion, then $f(\vec{x}_n)$ is actually defined by a certain set of equations involving g, h etc. Kleene [9] considers a more general class of systems of equations which can be used to define functions. The set of functions thus obtained is once again $\mathcal{P}$.

For Turing machines there are again a number of variants; they are obtained by allowing any combination of the following features: any number of tapes; tapes which may be infinite in either one or both directions; several tape heads on each tape; multi-dimensional tapes. In each case an equivalent model of computation is obtained. Part of this claim will be shown in (DCC).

The class of Turing machines may also be restricted in certain ways without restricting the set of T-computable functions. This is illustrated by the following

Exercise 4.15: (i) [3] Show that every task that can be performed by a TM (computing a partial recursive function, deciding a recursive set or accepting an r.e. set) is also performed by some TM whose only tape symbols are the input symbols and $\sqcup$. (You need not convert an arbitrary TM into one without extra symbols, although a solution on this basis is also possible. But the matter is in fact much simpler!)

(ii) [6] Show that every partial recursive function $f: (\Sigma^*)^n \to \Sigma^*$ and every recursive set $A \subseteq (\Sigma^*)^n$ is computed (decided) by some TM with only three states. Also, every r.e. set $A \subseteq (\Sigma^*)^n$ is accepted by some TM with only two states.

(iii) [1] Show that not every partial recursive function is computed by some TM which combines the conditions from (i) and (ii).

If TM's are used to accept sets rather than to compute functions, then yet another modification is possible, namely *nondeterminism.* A nondeterministic TM (NTM) $M = (Q, \Gamma, \delta, q_0, F)$ differs from a deterministic one (as considered so far) in the following way: for $q \in Q$, $a \in \Gamma$, $\delta(q, a)$ is no longer a triple (q', b, D), but a *finite set of* such triples, each of which indicates a possible next transition to be applied. Hence there may be several possible computations for every input $\vec{x}_n$. We say that M accepts $\vec{x}_n$, iff at least one computation for input $\vec{x}_n$ leads to termination in an accepting state.

Similarly, the phrase '$\mathcal{A}$ accepts L' is defined for any type of nondeterministic algorithm $\mathcal{A}$. One example are *nondeterministic Markov algorithms* (NMA's). An NMA M, like an MA, consists of a finite set of instructions of the form $u_i \to v_i$, but now a step of a computation is any step which transforms a string of the form xu_iy into xv_iy. It is no longer required that i is chosen as small as possible or that the *first* occurrence of u_i as a substring is replaced. M accepts $\vec{w}_n \in (\Sigma^*)^n$ iff at least one computation of M for input $\vec{w}_n$ terminates, that is, leads to a configuration where no instruction can be applied any more.

Other variants of the notion of Markov algorithm are also possible. For instance, Brainerd/Landweber [2] consider so-called *labelled* Markov algorithms.

NMA's are closely related to various other types of rewriting systems, namely *Post systems*, *semi-Thue systems*, *Thue systems*, and *unrestricted* (*type 0*) *grammars*.

A (*nondeterministic*) *Post system* (NPS) P over Γ is a set of instructions (u_i, v_i) with u_i, $v_i \in \Gamma^*$. This time the corresponding relation $\xrightarrow[P]{}$ on Γ^* is defined differently than for NMA's: $x \xrightarrow[P]{} y$ holds iff there exists an instruction (u_i, v_i) and a string $z \in \Gamma^*$ such that $x = u_iz$ and $y = zv_i$. If this is the case, then the transformation of x into y is a step of a corresponding computation. As for NMA's, $\vec{w}_n$ is accepted by P iff the computation of P starting with the string $\sqcup w_1 \ldots \sqcup w_n$ terminates, and the language accepted by P is the set of all strings accepted by P.

Thus we have considered three different classes of nondeterministic procedures. The class of languages accepted by some NTM, the class of languages accepted by some NMA, and the set of languages accepted by some NPS all coincide with the set of r.e. languages.

A *semi-Thue system* S over an alphabet Γ consists again of a finite set of instructions of the form $u_i \to v_i$ with u_i, $v_i \in \Gamma^*$, and a relation $\xrightarrow[S]{}$ is defined as for NMA's. However, S is not regarded as a procedure for accepting sets. Instead, one is interested in the induced relation $\xrightarrow[S]{*}$ on Γ^*. The *transformation problem* for semi-Thue systems is the problem of deciding for a given semi-Thue system S over Γ and given strings x,

$y \in \Gamma^*$ whether or not $x \xrightarrow[S]{*} y$ holds, that is, whether or not x can be transformed into y in a finite number of steps by the rewriting rules of S.

A *Thue system* is like a semi-Thue system, except that each rewriting rule may be applied in either direction. It may be viewed as a semi-Thue system such that whenever $u_i \to v_i$ is an instruction, then so is $v_i \to u_i$. It will be shown in (DCC) that the transformation problem for semi-Thue systems, and also for Thue systems, is undecidable. For Thue systems it is equivalent to the word problem for semigroups.

An *unrestricted (type 0) grammar* G is a quadruple (V, Σ, P, S) such that

(i) V and Σ are disjoint alphabets, the set of *metavariables* and the set of *terminals*.

(ii) P is a finite set of *productions* of the form $u_i \to v_i$ with $u_i \in (V \cup \Sigma)^+$ and $v_i \in (V \cup \Sigma)^*$.

(iii) S is an element of V, called the *start symbol*.

The set of productions P induces a relation $\xrightarrow[G]{}$ on $(V \cup \Sigma)^*$ as for NMA's and semi-Thue systems. The *language generated by* G is the set of all strings $w \in \Sigma^*$ such that $S \xrightarrow[G]{*} w$. It can be shown that the set of all languages generated by unrestricted grammars is once again the set of all r.e. languages.

We consider one more formalism for defining the class of partial recursive functions, unrelated to the ones dealt with before, namely λ*-calculus*. Its objects are certain expressions called λ*-terms*, which may be interpreted as functions. No distinction is made between functions and elements to which they are applied, similarly as in set theory, where every object is a set but also serves as an element for other sets. Among the λ-terms are countably many variables x, y, $\ldots$, and from λ-terms A, B and a variable x one can form new λ-terms AB ('application of A to argument B') and $\lambda x.A$ (λ-abstraction, interpreted as 'A, regarded as a function of x'). Functions of several arguments are avoided by stipulating that $\lambda x_1 x_2 \ldots x_n.A$ means $\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.A)\ldots)$. An occurrence of a variable x in a λ-term A is *bound*, if it appears within the range of some λ-abstraction $\lambda x.(\ldots)$; otherwise it is *free*.

There are two rules which allow to manipulate λ-terms. B is derived from A by an application of the α*-rule* ($A \xrightarrow[\alpha]{} B$), iff B is obtained from A by replacing a subterm of the form $\lambda x.C$ by $\lambda y.C[y/x]$, where y does not occur free in C; $C[y/x]$ denotes the result of substituting y for every free occurrence of x in C. This amounts to renaming a bound variable.

B is derived from A by an application of the β*-rule* ($A \xrightarrow[\beta]{} B$), iff B is obtained from A by replacing a subterm of the form $(\lambda x.C)D$ by $C[D/x]$. This corresponds to the intuitive meaning of applying the function $\lambda x.C$ to the argument D. However, this substitution must not cause any free variable of D to get bound by some λ-abstraction within C; to avoid this, bound variables have to be renamed if necessary. For instance, a correct application of the β-rule transforms $(\lambda xy.x)y$ not into $\lambda y.y$ (which would alter the intuitive meaning), but into $\lambda z.y$.

Let $A \xrightarrow[\lambda]{} B$ be the relation which holds iff A can be transformed into B by an application of the α- or β-rule. The reflexive, symmetric and transitive closure of $\xrightarrow[\lambda]{}$ is denoted by $=_\beta$. Thus $A =_\beta B$ holds iff there exists a sequence of λ-terms A_0, $\ldots$, A_n such that $A_0 = A$, $A_n = B$ and $A_{i-1} \xrightarrow[\lambda]{} A_i$ or $A_i \xrightarrow[\lambda]{} A_{i-1}$ holds for all i. It can be shown that this is equivalent to the existence of a λ-term C such that $A \xrightarrow[\lambda]{*} C$ and $B \xrightarrow[\lambda]{*} C$.

In order to specify λ-terms as defining functions $f \in \mathcal{NF}$, one first has to associate a λ-term $\overline{m}$ with every natural number n. Intuitively, $\overline{m}$ denotes the function which maps every function f to f^m. Thus

$$\begin{aligned} \overline{0} &= \lambda f x.x, \\ \overline{m+1} &= \lambda f x.f(\overline{m} f x). \end{aligned}$$

Note that by the above conventions $\lambda f x.x$ denotes the function whose value for every argument f is the identity function $\lambda x.x$, and $\lambda f x.f(\overline{m} f x)$ denotes the function which for argument f gives as its value the function $\lambda x.f\big(f^m(x)\big) = \lambda x.f^{m+1}(x)$. A λ-term F *λ-defines* a function $f\colon \omega^n \to \omega$, iff for all $\vec{x}_n$, y we have

$$f(\vec{x}_n) = y \iff F\overline{x}_1 \ldots \overline{x}_n =_\beta \overline{y}.$$

Then it can be shown that the λ-definable functions are exactly the partial recursive functions.

5. Undecidability, enumerations, and the recursion theorem

5.1. Undecidability.

We begin this section with two more theorems which can be shown by diagonalization. In the proof of theorem 3.7 (i) or (ii) a recursive function f was defined which is not primitive recursive; the construction was based on an enumeration $(\psi'_0, \psi'_1, \ldots)$ of $\mathcal{PR}^{(1)}$ for which the corresponding universal function was recursive. If a similar enumeration existed for $\mathcal{R}^{(1)}$, then an analogous argument would show that there exists a recursive function which is not recursive; this contradiction shows that no such enumeration exists. This is essentially the assertion of the following theorem.

Theorem 5.1: There exists no r.e. set A such that

$$\mathcal{R}^{(1)} = \{\, \phi_x \mid x \in A \,\}. \tag{1}$$

Proof: If $A = \emptyset$, then obviously A does not satisfy (1). Thus let A be a non-empty r.e. set. By theorem 3.8 (i) A is the range of a recursive function f. Now define

$$g(x) = \phi_{f(x)}(x) + 1 = \psi\big(f(x), x\big) + 1.$$

If $\phi_{f(x)}(x)\!\downarrow$ for every x, then $g(x) \neq \phi_{f(x)}(x)$, hence $g \neq \phi_{f(x)}$ for every x, hence g is a recursive function $\notin \{\, \phi_x \mid x \in A \,\}$. If $\phi_{f(x)}(x)\!\uparrow$ for some x, then the set $\{\, \phi_x \mid x \in A \,\}$ contains a function $\phi_{f(x)}$ which is not total, that is, not recursive. In both cases A fails to satisfy (1). $\square$

Corollary 5.2: The set (P) $\mathrm{Tot} = \{\, x \mid \phi_x \text{ is recursive} \,\}$ is not r.e. $\square$

Note that theorem 5.1 is stronger than its corollary. The latter does not rule out that there exists an r.e. set A such that A contains at least one index of every recursive function, and no index of a nonrecursive function.

Exercise 5.1: [3] Explain why theorem 5.1 is essentially equivalent to the assertion that there exists no enumeration of $\mathcal{R}^{(1)}$ for which the corresponding universal function is recursive.

We have seen that by 'diagonalizing out of' the set $\mathcal{PR}^{(1)}$ or $\mathcal{R}^{(1)}$ one obtains a recursive function which is not primitive recursive, or a total nonrecursive function respectively. Now, what happens if diagonalization is applied to the set $\mathcal{P}^{(1)}$? One might suppose that the result is a function which is not partial recursive, yet, if one uses the enumeration $(\phi_0, \phi_1, \ldots)$ and proceeds as for $\mathcal{PR}^{(1)}$ or $\mathcal{R}^{(1)}$, then one obtains a function f like $\lambda x.\phi_x(x) + 1$ or $\lambda x.1 \dot{-} \phi_x(x)$, which *is* partial recursive. Why does this not lead to a contradiction? The answer is that this definition of f does *not* guarantee $f(x) \neq \phi_x(x)$, that is, $f \neq \phi_x$ for all x, since $f(x)$ and $\phi_x(x)$ may both be undefined. Nevertheless, it turns out that such a function f satisfies an interesting property.

Theorem 5.3: There exists a partial recursive function $f\colon\omega \to \omega$ with range $\{0, 1\}$ which has no recursive extension.

Proof: Let $f(x) = 1 \dot{-} \phi_x(x)$, as above. Then no ϕ_x can be a recursive extension of f, for if ϕ_x is total, then $\phi_x(x) \neq f(x) = 1 \dot{-} \phi_x(x)$. □

Now we introduce the second basic method for showing undecidability results, namely *reduction*. Roughly speaking, reducing a problem P_1 to another problem P_2 amounts to showing that an algorithm for solving P_2 also yields an algorithm for P_1. Such a result may occasionally be used to show that P_1 is algorithmically solvable, but more important are inferences in the opposite direction: If P_1 is known to be undecidable, then the same can be deduced for P_2. This method is quite powerful and will be used shortly to prove a general theorem which asserts the undecidability of a large class of sets. But first we illustrate the method by an example.

Example 5.1: It follows from theorem 5.1 that the set $\mathrm{Tot} = \{\, x \mid \phi_x \text{ is total} \,\}$ is not recursive; we now provide a different proof. Applying corollary 3.5 to the function $g(x, y) = \phi_x(x)$ yields a recursive function f such that for all x, y

$$\phi_{f(x)}(y) = \phi_x(x).$$

Then for all x we have

$$x \in K \iff \phi_x(x)\!\downarrow \iff \phi_{f(x)}(y)\!\downarrow \text{ for all } y \iff f(x) \in \mathrm{Tot}.$$

Hence Tot recursive would imply that K is also recursive, a contradiction.

In this example we provided a reduction of (the decision problem for) K to (the decision problem for) Tot of the form given by the following

Definition 5.1: Let A, $B \subseteq \omega$. Then A is *m-reducible* to B ($A \leq_m B$) (via f), iff the function $f\colon \omega \to \omega$ is recursive, and

$$(\forall x)(x \in A \iff f(x) \in B).$$

If in addition f is one-one, then A is *1-reducible* to B ($A \leq_1 B$) (via f).

In the present chapter these two reducibilities play essentially the same rôle. In chapter 6 yet another reducibility relation will be defined, and all three will be investigated more systematically. The following two theorems summarize a few very simple facts.

Theorem 5.4: Let $\leq_r$ be either $\leq_m$ or $\leq_1$. Then for all A, B, $C \subseteq \omega$ we have

(i) $A \leq_r B \iff \overline{A} \leq_r \overline{B}$.

(ii) $A \leq_r A$.

(iii) $A \leq_r B \;\&\; B \leq_r C \Longrightarrow A \leq_r C$.

Proof: (i) It follows immediately from the definition that $A \leq_r B$ via f implies $\overline{A} \leq_r \overline{B}$ via f and vice versa.

(ii) Clearly, $A \leq_r A$ via $\lambda x.x$.

(iii) If $A \leq_r B$ via f and $B \leq_r C$ via g, then $A \leq_r C$ via gf, since for all x

$$x \in A \iff f(x) \in B \iff gf(x) \in C.$$

□

Theorem 5.5: Let $A \leq_m B$. Then

(i) If B is recursive, then A is recursive.

(ii) If B is r.e., then A is r.e.

A fortiori, the same conclusions follow under the condition $A \leq_1 B$.

Proof: If $A \leq_m B$ via f and B is recursive, then $(\forall x)\big(x \in A \iff f(x) \in B\big)$ implies that A is recursive by theorem 2.4 (ii). Part (ii) is a special case of theorem 3.10. □

Part (ii) of this theorem indicates that the method from example 5.1 can also be used to show that certain sets are not r.e.; this will be illustrated by example 5.2.

Notation: Here are the definitions of certain sets which will be used in the following examples; K_1 and Tot are also given again:

$$\begin{aligned}
K_1 &= \{\, (x,y,z) \mid \phi_x(y) = z \,\}, \\
\mathrm{Empt} &= \{\, x \mid W_x = \emptyset \,\}, \\
\mathrm{Tot} &= \{\, x \mid \phi_x \text{ is total} \,\}, \\
\mathrm{Const} &= \{\, x \mid \phi_x \text{ is total and constant} \,\}, \\
\mathrm{Prim} &= \{\, x \mid \phi_x \text{ is primitive recursive} \,\}, \\
\mathrm{Fin} &= \{\, x \mid W_x \text{ is finite} \,\}, \\
\mathrm{Inf} &= \{\, x \mid W_x \text{ is infinite} \,\}, \\
\mathrm{Cof} &= \{\, x \mid W_x \text{ is cofinite, that is, } \overline{W_x} \text{ is finite} \,\}, \\
\mathrm{Rec} &= \{\, x \mid W_x \text{ ist recursiv} \,\}, \\
\mathrm{Ext} &= \{\, x \mid \phi_x \text{ has a recursive extension} \,\}.
\end{aligned}$$

Example 5.2: We show that among the sets just listed and their complements none is recursive, and only K_1 and $\overline{\mathrm{Empt}}$ are r.e. The fact that K_1 is r.e. was shown in example 3.2 (ii); for $\overline{\mathrm{Empt}}$ this follows from the representation

$$\overline{\mathrm{Empt}} = \{\, x \mid (\exists y, u)\big(T(u,x,y)\big) \,\}.$$

Now consider once again the function f from example 5.1. It was defined such that $\phi_{f(x)}$ is

$$\begin{cases} \text{total} & \text{if } x \in K, \\ \text{nowhere defined} & \text{if } x \notin K. \end{cases} \tag{1}$$

The function f is one-one, since it was obtained by an application of corollary 3.5, and (1) actually implies that the following relations all hold via f:

$$\begin{aligned}
\overline{K} &\leq_1 \mathrm{Empt}, \\
\overline{K} &\leq_1 \overline{\mathrm{Tot}}, \\
\overline{K} &\leq_1 \mathrm{Fin} = \overline{\mathrm{Inf}}, \\
\overline{K} &\leq_1 \overline{\mathrm{Cof}}.
\end{aligned}$$

Since $\overline{K}$ is not r.e. (theorem 3.15), we conclude that Empt, $\overline{\mathrm{Tot}}$, Fin $= \overline{\mathrm{Inf}}$ and $\overline{\mathrm{Cof}}$ are not r.e., hence Empt, Tot, Fin, Inf and Cof are not recursive. By a slight modification of f it can be shown that $\overline{K_1}$, $\overline{\mathrm{Const}}$ and $\overline{\mathrm{Prim}}$ are not r.e. either, see exercise 5.2 (i), (ii).

A new idea is needed to show that Tot, Const, Prim, $\overline{\text{Fin}}$ = Inf and Cof are not r.e. We wish to construct f_1 such that $\overline{K} \leq_1$ Tot via f_1. Applying corollary 3.5 again, we have $\phi_{f_1(x)}(y) = g_1(x, y)$, where g_1 should be defined such that $g_1(x, y)\downarrow$ for all y iff $x \notin K$. Intuitively, $g_1(x, y)$ is computed as follows: carry out y steps of the computation of $\phi_x(x)$ and output 0 iff this computation does *not* terminate, otherwise give no output. Then $g_1(x, y) = 0$ for all y if $x \notin K$, but $g_1(x, y)\uparrow$ for sufficiently large y if $x \in K$. Formally, define

$$g_1(x, y) = \phi_{f_1(x)}(y) = \begin{cases} 0 & \text{if } (\forall z < y)\big(\neg T(z, x, x)\big), \\ \uparrow & \text{otherwise.} \end{cases}$$

Then

$$\phi_{f_1(x)} \begin{cases} \text{has finite domain} & \text{if } x \in K, \\ \text{equals } \lambda x.0 & \text{if } x \notin K. \end{cases}$$

Hence the following relations hold via f_1:

$$\begin{aligned} &\overline{K} \leq_1 \text{Tot}, \\ &\overline{K} \leq_1 \text{Const}, \\ &\overline{K} \leq_1 \text{Prim}, \\ &\overline{K} \leq_1 \overline{\text{Fin}} = \text{Inf}, \\ &\overline{K} \leq_1 \text{Cof}. \end{aligned}$$

Thus Tot, Const, Prim, $\overline{\text{Fin}}$ = Inf and Cof are not r.e., and therefore not recursive.

Now it only remains to deal with the sets Rec and Ext and their complements. This is done in part (iii) of the following exercise.

Exercise 5.2: (i) [1] Modify the function f from example 5.1 to construct f_0 such that $\overline{K} \leq_1 \overline{\text{Const}}$ and $\overline{K} \leq_1 \overline{\text{Prim}}$ via f_0.

(ii) [2] Show that $\overline{K_1}$ is not r.e.

(iii) [4] Construct a function f_2 such that Tot $\leq_1$ Inf, Tot $\leq_1$ $\overline{\text{Rec}}$ and Tot $\leq_1$ $\overline{\text{Ext}}$ all hold via f_2. Conclude that the sets Rec, $\overline{\text{Rec}}$, Ext and $\overline{\text{Ext}}$ are not r.e.

In the rest of this exercise prove the following further reducibility relations:

(iv) [3] Inf $\leq_1$ Tot.

(v) [4] Tot $\leq_1$ Rec.

(vi) [6] Cof $\leq_1$ Rec. (Here you are allowed to use ChT.)

Exercise 5.3: [8] Show that Fin $\not\leq_m$ Tot.

In (DCC) we will be in a position to provide a complete classification of sets like Tot, Fin etc. with respect to reducibilities, when the *arthmetical hierarchy* is introduced. At the present stage non-reducibility results, like that of exercise 5.3, would be rather difficult to show. We can only deduce $A \not\leq_m K$ for A not r.e., using theorem 5.5 (ii).

Several of the sets just considered are of the following type:

Definition 5.2: A set $A \subseteq \omega$ is a *function index set*, iff A is of the form

$$\{ x \mid \phi_x^{(n)} \in \mathcal{C} \},$$

where $\mathcal{C} \subseteq \mathcal{P}^{(n)}$ for some n. A is a *set index set*, iff A is of the form

$$\{ x \mid W_x^{(n)} \in \mathcal{W} \},$$

where $\mathcal{W} \subseteq \mathcal{E}^{(n)}$. A is *trivial*, iff $A = \emptyset$ or $A = \omega$. (In this case A is trivially both a function- and a set index set.)

Function index sets A are characterized by the following property: If $x \in A$, then A also contains all other indices of $\phi_x^{(n)}$, with n as above. Set index sets are characterized similarly, with $W_x^{(n)}$ instead of $\phi_x^{(n)}$. Every set index set is also a function index set, since

$$\{ x \mid W_x^{(n)} \in \mathcal{W} \} = \big\{ x \mid \phi_x^{(n)} \in \{ f \in \mathcal{P}^{(n)} \mid \mathrm{dom}(f) \in \mathcal{W} \} \big\}.$$

Exercise 5.4: [4] Which of the sets K, Empt, Tot, Const, Prim, Fin, Inf, Cof, Rec and Ext are of the form $\{ x \mid \phi_x \in \mathcal{C} \}$ ($\mathcal{C} \subset \mathcal{P}^{(1)}$), and which are of the form $\{ x \mid W_x \in \mathcal{W} \}$ ($\mathcal{W} \subset \mathcal{E}^{(1)}$)? (Of these sets only K is somewhat tricky to deal with. For those sets which are not of the form $\{ x \mid \phi_x \in \mathcal{C} \}$ or $\{ x \mid W_x \in \mathcal{W} \}$ respectively, you need not worry about whether they are function- (set-) index sets after all, being of the form $\{ x \mid \phi_x^{(n)} \in \mathcal{C} \}$ or $\{ x \mid W_x^{(n)} \in \mathcal{W} \}$ with $\mathcal{C} \subset \mathcal{P}^{(n)}$ or $\mathcal{W} \subset \mathcal{E}^{(n)}$ for some $n > 1$. Actually, it is not very difficult to see that the answer is negative in each case, but this depends on our specific enumerations $(\phi_0^{(n)}, \phi_1^{(n)}, \ldots)$, and it is of no importance whatsoever.)

The main result of the present chapter is the following:

Theorem 5.6 (Rice-theorem): If A is a non-trivial function index set, then A is not recursive.

Proof: Let $A = \{ x \mid \phi_x^{(n)} \in \mathcal{C} \}$, $\mathcal{C} \subset \mathcal{P}^{(n)}$, A non-trivial. Without loss of generality assume that $\mathcal{C}$ does not contain the nowhere defined function $\lambda \vec{x}_n.\uparrow$, otherwise replace $\mathcal{C}$ by $\overline{\mathcal{C}}$ (the complement of $\mathcal{C}$ with respect to $\mathcal{P}^{(n)}$); this amounts to considering $\overline{A}$ instead of A, and $\overline{A}$ is recursive iff A is recursive. Also, let h be an element of $\mathcal{C}$; such an element exists, otherwise $\mathcal{C} = \emptyset$, and $A = \emptyset$ would be trivial. Now let f be recursive such that for all x, $\vec{y}_n$

$$\phi_{f(x)}^{(n)}(\vec{y}_n) = \begin{cases} h(\vec{y}_n) & \text{if } x \in K, \\ \uparrow & \text{if } x \notin K. \end{cases}$$

(This involves an application of theorem 3.12, besides corollary 3.5; the right-hand side of this equation is a partial recursive function of x and y.) Then

$$\phi_{f(x)}^{(n)} = \begin{cases} h \in \mathcal{C} & \text{if } x \in K, \\ \lambda \vec{y}_n.\uparrow \notin \mathcal{C} & \text{if } x \notin K. \end{cases}$$

Therefore $K \leq_m A$ via f; in fact $K \leq_1 A$ via f, if the function f is constructed in the usual way. Hence A is not recursive. □

Theorem 5.6 and exercise 5.4 allow to deduce immediately all undecidability results from example 5.2, but they do not allow to deduce that certain sets are not r.e. It is natural to search for a criterion for r.e. index sets. Such a criterion will be given (theorem 5.18) after further preparations.

5.2. Indices for sets and uniform transformations.

So far two kinds of indices for subsets of ω have been introduced: canonical indices for finite sets (definition 2.14), and r.e. indices for r.e. sets (definition 3.4). A third kind of index is suitable for *recursive* sets:

Definition 5.3: A number u is called a *characteristic index* of a set $A \subseteq \omega^n$, iff $c_A = \phi_u^{(n)}$; in this case A is denoted by $C_u^{(n)}$. (If ϕ_u is not a characteristic function, then $C_u^{(n)}$ has no meaning.) In the case $n = 1$ we write C_u instead of $C_u^{(1)}$.

Of course it follows from theorem 3.6 that every recursive set has infinitely many characteristic indices, just as every r.e. set has infinitely many r.e. indices.

For finite sets A all three kinds of indices exist. It is possible to effectively compute a characteristic index of A from the canonical index, or an r.e. index from a characteristic index of A, but the converse fails in both cases. This is the contents of the following two theorems:

Theorem 5.7: There exist primitive recursive functions f, g such that for all x

(i) $D_x = C_{f(x)}$;

(ii) If ϕ_x is a characteristic function, then $C_x = W_{g(x)}$.

Proof: (i) Using exercise 2.18 (i), define f such that for all x, y

$$\phi_{f(x)}(y) = \begin{cases} 1 & \text{if } y \in D_x, \\ 0 & \text{if } y \notin D_x. \end{cases}$$

(ii) Define g such that for all x, y

$$\phi_{g(x)}(y) = \left\{ \begin{array}{ll} 0 & \text{if } \phi_x(y) = 1, \\ \uparrow & \text{otherwise} \end{array} \right\} = g_1(\phi_x(y)),$$

where

$$g_1(z) = \begin{cases} 0 & \text{if } z = 1, \\ \uparrow & \text{if } z \neq 1. \end{cases}$$

□

Theorem 5.8: (i) There exists no partial recursive function f with the following property: If ϕ_x is a characteristc function and C_x is finite, then $C_x = D_{f(x)}$.

(ii) There exists no partial recursive function f with the following property: If W_x is finite, then $W_x = C_{f(x)}$. (Hence, a fortiori, no partial recursive function f satisfies $W_x = C_{f(x)}$ for every *recursive* W_x.)

Proof: (i) Suppose f is a partial recursive function which satisfies $C_x = D_{f(x)}$ whenever C_x is defined and finite. Let g be recursive such that for all x, y

$$\phi_{g(x)}(y) = \begin{cases} 1 & \text{if } T(y, x, x), \\ 0 & \text{if } \neg T(y, x, x). \end{cases}$$

Then $\phi_{g(x)}$ is a characteristic function for all x, and

$$C_{g(x)} \begin{cases} \text{contains exactly one element} & \text{if } x \in K, \\ = \emptyset & \text{if } x \notin K. \end{cases}$$

Hence for all x, $fg(x)\downarrow$ and

$$x \in K \iff C_{g(x)} \neq \emptyset \iff D_{fg(x)} \neq \emptyset \iff fg(x) \neq 0.$$

This would imply that K is recursive, a contradiction. Therefore no partial recursive function satisfies the assumed property.

(ii) Assume f is a partial recursive function such that $W_x = C_{f(x)}$ for W_x finite. Let g be recursive such that for all x, y

$$\phi_{g(x)}(y) = \begin{cases} \phi_x(x) & \text{if } y = 0, \\ \uparrow & \text{if } y > 0. \end{cases}$$

Then for all x, $W_{g(x)} \subseteq \{0\}$, $fg(x)\downarrow$, and

$$x \in K \iff 0 \in W_{g(x)} \iff 0 \in C_{fg(x)} \iff \phi_{fg(x)}(0) = 1.$$

As in (i) this would imply that K is recursive, a contradiction. □

Theorems 5.7 and 5.8 can be interpreted as follows: A canonical index for a finite set provides more information than a characteristic index, and a characteristic index for a recursive set provides more information than an r.e. index. Indeed, if a canonical index for a finite set is known, then the set itself is known completely, and one can effectively compute a list of all its elements. If a characteristic index is known, then one can still effectively decide for any given element whether or not it belongs to the set, but one does not have a complete overview over all its elements; for instance, one does not necessarily know the number of elements, even if the set is known to be finite (see the proof of theorem 5.8 (i).) Finally, an r.e. index provides even less information then a characteristic index: It allows to generate the set, but it does not provide a method for deciding membership for a given element.

Example 5.3: Let A be the set of all $x \in \omega$ which cannot be represented as a sum of 16 bisquares (fourth powers of natural numbers). It is easy to write a program for deciding $x \in A$ for any given x; translating such a program into $\mathcal{WL}$ and computing its code number yields a characteristic index for A. Also, it has been shown that A is finite. On the other hand, a complete list of elements of A, and the cardinality of A, are unknown. Thus, based on current knowledge, it is clear how to determine a characteristic index, but not how to determine a canonical index of A.

We already proved a number of results which state that $\mathcal{R}_*$ and $\mathcal{E}_*$ are closed under various operations. See for instance, examples 2.3 (iii) and 2.4 (v), theorems 2.9, 3.9 and 3.10, as well as exercise 3.13. In the rest of this section more facts of this kind will be shown, and as an additional piece of information it will be proven that most of these closure properties hold *uniformly*. For instance, the class of r.e. sets is closed under union *uniformly for r.e. indices*; this means that an r.e. index for $A \cup B$ can be computed effectively from r.e. indices of A and B. The results of exercise 2.18 assert that $A^\#$ is closed under various operations uniformly for canonical indices.

Theorem 5.9: There exist primitive recursive functions f, g, h, f', g' such that the following holds for all x, y: If ϕ_x and ϕ_y are characteristic functions, then

$$\begin{aligned}
C_{f(x,y)} &= C_x \cup C_y, \\
C_{g(x,y)} &= C_x \cap C_y, \\
C_{h(x)} &= \overline{C_x}, \\
C_{f'(x,y)} &= C_x - C_y.
\end{aligned}$$

Also, if ϕ_x is a characteristic function and ϕ_y is recursive, then

$$C_{g'(x,y)} = \phi_y^{-1}(C_x).$$

Proof: Define f, g etc. such that for all x, y, z

$$\begin{aligned}
\phi_{f(x,y)}(z) &= \mathrm{sig}\big(\phi_x(z) + \phi_y(z)\big), \\
\phi_{g(x,y)}(z) &= \phi_x(z) \cdot \phi_y(z), \\
\phi_{h(x)}(z) &= 1 \dot{-} \phi_x(z), \\
\phi_{f'(x,y)}(z) &= \phi_x(z) \dot{-} \phi_y(z), \\
\phi_{g'(x,y)}(z) &= \phi_x\big(\phi_y(z)\big).
\end{aligned}$$

□

Theorem 5.10: There exist primitive recursive functions f, g, h, f' such that for all x, y

$$\begin{aligned}
W_{f(x,y)} &= W_x \cup W_y, \\
W_{g(x,y)} &= W_x \cap W_y, \\
W_{h(x,y)} &= \phi_y^{-1}(W_x), \\
W_{f'(x,y)} &= \phi_y(W_x).
\end{aligned}$$

Proof: Intuitively, if x and y are given, then an enumeration of $W_x \cup W_y$ is obtained by simultaneously carrying out the enumeration of W_x and W_y and outputting those elements which appear in at least one of these enumerations. If ChT is accepted, then this argument already allows to conclude the existence of a function f with the required property as follows: By performing the enumeration described simultaneously for all x and y, one obtains a method for generating $A = \{\, (x, y, z) \mid z \in W_x \cup W_y \,\}$, hence A is r.e., and there exists a partial recursive function f_0 with domain A. Then the function f defined by $\phi_{f(x,y)}(z) = f_0(x, y, z)$ satisfies the required condition.

Informal arguments for the existence of desired functions g, h and f' proceed similarly. A formal proof also applies corollary 3.5, but to a function which can be defined readily without ChT:

$$\begin{aligned}
\phi_{f(x,y)}(z) &= (\mu u)[T(u, x, z) \vee T(u, y, z)], \\
\phi_{g(x,y)}(z) &= \phi_x(z) + \phi_y(z).
\end{aligned}$$

Then $\phi_{f(x,y)}(z){\downarrow}$ iff $\phi_x(z){\downarrow}$ & $\phi_y(z){\downarrow}$, and $\phi_{g(x,y)}{\downarrow}$ iff $\phi_x(z){\downarrow} \vee \phi_y(z){\downarrow}$, so the required properties are satisfied.

The formal definition of h and f' again uses the Kleene predicate in an appropriate way; this is left for the following exercise. □

Exercise 5.5: [2] (i) Complete the formal proof of theorem 5.10.
(ii) Prove also that there exist primitive recursive functions f_1, f_2, ..., f_8 such that

$$\begin{aligned}
W_{f_1(x)} &= \{ z \mid (\exists y)\big((y,z) \in W_x^{(2)}\big) \},\\
W_{f_2(x,u)} &= \{ z \mid (\exists y < u)\big((y,z) \in W_x^{(2)}\big) \},\\
W_{f_3(x,u)} &= \{ z \mid (\forall y < u)\big((y,z) \in W_x^{(2)}\big) \},\\
W_{f_4(x,u)} &= \{ z \mid (\exists y \in D_u)\big((y,z) \in W_x^{(2)}\big) \},\\
W_{f_5(x,u)} &= \{ z \mid (\forall y \in D_u)\big((y,z) \in W_x^{(2)}\big) \},\\
W_{f_6(x)} &= \{ y \mid D_y \cap W_x \neq \emptyset \},\\
W_{f_7(x)} &= \{ y \mid D_y \subseteq W_x \},\\
W_{f_8(x)} &= \bigcup_{y \in W_x} W_y.
\end{aligned}$$

Many further uniformity results follow trivially by combining the results presented. For instance, the class of recursive sets is also closed under union or intersection uniformly for *r.e.* indices, using the same functions f, g as in theorem 5.10. As another example, there exists a recursive function h such that for all x, y we have $W_{h(x)} = \phi_y(C_x)$ if ϕ_x is a characteristic function. Just define $h(x,y) = f'\big(g(x), y\big)$, where g is as in theorem 5.7 and f' is as in theorem 5.10.

Of course, the last two theorems should not suggest that $\mathcal{R}_*$ and $\mathcal{E}$ are closed under every operation conceivable, or that every closure property is uniform. Actually, in both theorems one closure property was not considered, and the missing one is in fact not satisfied. Indeed, we know that K is an example of an r.e. set whose complement is not r.e. The next theorem shows that an r.e. index of $\overline{W_x}$ cannot be computed effectively from x, even if W_x is recursive:

Theorem 5.11: There exists no partial recursive function f such that whenever W_x is recursive, then $f(x)\!\downarrow$ and $W_{f(x)} = \overline{W_x}$.

Proof: Suppose $f \in \mathcal{P}$ satisfies the property of the theorem. Let g be recursive such that for all x, y

$$\phi_{g(x)}(y) = \phi_x(x).$$

Then for all x, $W_{g(x)}$ is recursive (either $\emptyset$ or ω), hence $fg(x)\!\downarrow$, that is, fg is recursive, and

$$x \in \overline{K} \iff 0 \notin W_{g(x)} \iff 0 \in W_{fg(x)} \iff (\exists z, u)\big(z = fg(x) \,\&\, T(u, z, 0)\big).$$

It would follow by theorem 3.8 (iv) that $\overline{K}$ is r.e., a contradiction. Hence no such function f exists. □

Likewise, we know that $\mathcal{R}_*^{(1)}$ is not closed under the operation of taking images under partial recursive functions. In fact, $f(A)$ need not even be recursive, if $A = \omega$ and f is primitive recursive, since by exercise 3.12 (viii) every r.e. set except $\emptyset$ is of the form $f(\omega)$. Also, if A is recursive and f is partial recursive (rather than recursive), then $f^{-1}(A)$ is not necessarily recursive, as can be seen from the following counterexample:

Example 5.4: Let

$$f(x) = \left\{ \begin{matrix} x & \text{if } x \in K, \\ \uparrow & \text{if } x \notin K \end{matrix} \right\} = (\langle x, \phi_x(x) \rangle).0.$$

Then $f^{-1}(\omega) = K$.

Exercise 5.6: (i) [2] Let A be any fixed nonempty subset of ω. Show that no partial recursive function f satisfies

$$C_{f(x)} = \phi_x^{-1}(A)$$

whenever $\phi_x^{-1}(A)$ is recursive.

(ii) [4] Let A be as in (i). Show that no partial recursive function f satisfies

$$D_{f(x)} = \phi_x^{-1}(A)$$

whenever ϕ_x is recursive and one-one. (This assertion is nontrivial for finite A, since in this case $\phi_x^{-1}(A)$ is finite if ϕ_x is one-one.)

(iii) [4] Let A be a fixed *infinite* subset of ω. Show that no partial recursive function f satisfies

$$C_{f(x)} = \phi_x(A)$$

whenever ϕ_x is recursive and $\phi_x(A)$ is recursive.

(iv) [0] Show that there exists a partial recursive function f such that for all x, y with ϕ_y recursive,

$$D_{f(x,y)} = \phi_y(D_x).$$

(Parts (ii) and (iv) show that finite sets given by canonical indices behave in a sense opposite to recursive sets given by characteristic indices: They are uniformly closed under images under recursive functions, but not under inverse images under recursive functions.)

There are many other transformations for which the question of uniformity can be raised. The set of partial recursive functions is closed uniformly under many operations. This can usually shown by a trivial application of the S-m-n-theorem, as illustrated by example 3.1. As another example, consider theorem 3.8; again we restrict ourselves to the case $n = 1$. The proof shows that the transition from (ii) to (iii) is uniform in the following sense: there exists a primitive recursive function f such that for all x

$$\mathrm{dom}\big(\phi_{f(x)}\big) = \mathrm{ran}(\phi_x).$$

Similarly, it can be shown that all transitions between those representations of r.e. sets are uniform which are given in parts (ii), (iii), (iv), (v), (vi) and (vii) of theorem 3.8 and exercise 3.12 (for ordinary indices of partial recursive functions and characteristic indices of recursive sets). The same is true for the transition from (i) to (ii) in the following sense: Given the information that $A = \emptyset$, or an index x such that ϕ_x is recursive and $A = \mathrm{ran}(\phi_x)$, one can effectively find an index y such that $A = \mathrm{ran}(\phi_y)$, where ϕ_y need no longer be recursive. This can be expressed by saying that there exists a primitive recursive function f such that

$$\mathrm{ran}(\phi_{f(x)}) = \begin{cases} \emptyset & \text{if } x = 0, \\ \mathrm{ran}(\phi_{x \dot{-} 1}) & \text{if } x > 0. \end{cases}$$

On the other hand, the transition from (ii) to (i) is non-uniform: Given x such that $A = \mathrm{ran}(\phi_x)$, there is no effective decision procedure to decide whether $A = \emptyset$ or A is the range of a recursive function.

5.3. Enumerations.

In the first part of this section more results on the enumeration of subsets of ω are presented, thus supplementing theorem 3.8 (i), (ii). The following theorem shows that every r.e. set $A \subseteq \omega$ can be represented as the range of a partial recursive function g of a special form.

Definition 5.4: An *initial segment* of ω is a set of the form $\{\, x \mid x < a \,\}$ $(a \in \omega)$.

Theorem 5.12: There exists a recursive function f such that for all x

(i) $\mathrm{ran}(\phi_{f(x)}) = W_x$,
(ii) $\phi_{f(x)}$ is one-one, and
(iii) The domain of $\phi_{f(x)}$ is ω or an initial segment of ω.

Proof: Intuitively, $\phi_{f(x)}(y)$ is the first element found in an enumeration of W_x which differs from $\phi_{f(x)}(z)$ for all $z < y$; if no such element is found, then $\phi_{f(x)}(z)$ will be undefined for all $z \geq y$. Formally, define f such that $\phi_{f(x)}(y) = h(y, x)$, where

$$h(y, x) = \big((\mu u)[T(u.0, x, u.1) \,\&\, u.0.1 \notin \{h(0, x), \ldots, h(y-1, x)\}]\big).0.1.$$

Clearly, this definition of h can be represented as an instance of course-of-values recursion. The condition $T(u.0, x, u.1)$ implies that $\phi_x(u.1) = u.0.1$, hence $h(y, x) = u.0.1 \in W_x$, and the additional condition $u.0.1 \notin \{h(0, x), \ldots, h(y-1, x)\}$ guarantees that $\lambda y.h(y, x)$ is one-one. The value $h(y, x)$ is defined iff $h(z, x)$ is defined for all $z < y$ and W_x contains another element besides $h(0, x)$, …, $h(y-1, x)$.

It remains to show that $W_x \subseteq \mathrm{ran}(\phi_{f(x)})$. For any $z \in W_x$, $u = u_z$ be minimal such that $T(u.0, x, u.1)$ holds and $z = u.0.1$. Also, let $\{z_0, z_1, \ldots\}$ be an enumeration of W_x such that $u_{z_0} < u_{z_1} < u_{z_2} < \cdots$. Then the definition of $h(y, x) = \phi_{f(x)}(y)$ shows readily that $\phi_{f(x)}(y) = z_y$, proving the claim.

Exercise 5.7: [4] Prove the following generalization of theorem 5.3: If A is an infinite r.e. set, then there exists a partial recursive function f such that $\mathrm{dom}(f) \subset A$ and no partial recursive extension g of f satisfies $\mathrm{dom}(g) \supseteq A$.

The next result provides a characterization of $\mathcal{R}_*^{(1)}$ as a subset of $\mathcal{E}^{(1)}$, based on the representation of r.e. sets as ranges of recursive functions:

Theorem 5.13: (i) For every $A \subseteq \omega$, the following assertions are equivalent:

(a) A is recursive and $A \neq \emptyset$;
(b) There exists a nondecreasing recursive function f such that $A = \mathrm{ran}(f)$.

(ii) The following assertions are equivalent:

(c) A is recursive and infinite;
(d) A is infinite and $A = \mathrm{ran}(f)$ for some nondecreasing recursive function f;
(e) $A = \mathrm{ran}(f)$ for some increasing recursive function f.

Proof: Assume first that A is recursive. If A is finite, $A \neq \emptyset$, $A = \{a_0, \ldots, a_k\}$ with $a_0 < \cdots < a_k$, then $A = \mathrm{ran}(f)$, where

$$f(x) = \begin{cases} a_0 & \text{for } x = 0, \\ \vdots & \\ a_{k-1} & \text{for } x = k-1, \\ a_k & \text{for } x \geq k. \end{cases}$$

If A is infinite and a is the smallest element of A, then $A = \mathrm{ran}(f)$, where f is defined as follows:

$$f(0) = a,$$
$$f(x+1) = (\mu y)[y \in A \;\&\; y > f(x)].$$

Since the predicate $\lambda y.y \in A$ is recursive, so is function f. Also, f is increasing. This proves the implications (a) $\Longrightarrow$ (b) and (c) $\Longrightarrow$ (e); the implication (e) $\Longrightarrow$ (d) is trivial.

Conversely, let f be an nondecreasing recursive function, and let $A = \mathrm{ran}(f)$. If A is finite, then A is recursive (exercise 2.6 (i)), and we are done. If A is infinite, let

$$g(x) = (\mu z)[f(z) \geq x].$$

Then g is recursive, and for every x we have

$$x \in A \iff fg(x) = x.$$

Hence A is recursive. This proves the implications (b) $\Longrightarrow$ (a) and (d) $\Longrightarrow$ (c). □

Definition 5.5: Let $A \subseteq \omega$, and let $A = \{a_0, a_1, \ldots\}$ be the enumeration of A *in increasing order*. The *principal function* of A is the function (P) p_A defined by $p_A(x) = a_x$. (If A is finite, say $|A| = n$, then $p_A(x)\uparrow$ for $x \geq n$.)

Thus part (ii) of theorem 5.13 says that A is recursive if and only if p_A is recursive.

Exercise 5.8: (i) [4] Let $A \subseteq \omega$ be infinite. Show that if f and g are functions from ω to ω with range A, and f is one-one, then g does not dominate f. (This part has nothing to do with computability theory, it serves only as a preparation for (ii). The assertion is intuitively obvious, but a rigorous proof is slightly trickier than it may appear at first glance.)

(ii) [2] Use part (i) to show that exercise 3.12 (viii) and theorem 5.13 cannot be combined: There exists an r.e. (in fact, a recursive) set A such that no one-one primitive recursive function f has range A.

(iii) [4] Show that there exists a primitive recursive set A such that p_A dominates every primitive recursive function. Thus the set A from (ii) can even be made primitive recursive.

Theorem 5.14: Every infinite r.e. set A contains an infinite recursive subset.

Proof: Let f be a recursive function such that $\mathrm{ran}(f) = A$. Define g as follows:

$$g(0) = f(0),$$
$$g(x+1) = f\big((\mu z)[f(z) > g(x)]\big).$$

Then g is an increasing recursive function, and $\mathrm{ran}(g) \subseteq \mathrm{ran}(f)$. Thus, by theorem 5.13 (ii), $\mathrm{ran}(g)$ is an infinite recursive subset of A. □

Now we consider enumerations of *classes of sets*, which, naturally, amount to enumerations of corresponding sets of indices:

Definition 5.6: A set $\mathcal{C}$ of r.e. (finite) subsets of ω is *recursively* (*canonically*) *enumerable*, iff there exists an r.e. set A such that $\mathcal{C} = \{ W_x \mid x \in A \}$ ($\mathcal{C} = \{ D_x \mid x \in A \}$). A set $\mathcal{C}$ of recursive subsets of ω is *characteristically enumerable*, iff there exists an r.e. set A such that ϕ_x is a characteristic function for all $x \in A$ and $\mathcal{C} = \{ C_x \mid x \in A \}$. Finally, $\mathcal{C}$ is *completely recursively enumerable*, iff the set index set corresponding to $\mathcal{C}$, namely $\{ x \mid W_x \in \mathcal{C} \}$, is r.e.

Remark 5.1: The terminology of definition 5.6 is standard, but it is not compatible with definition 2.8 for the coding scheme $\mathrm{cd}_{\omega^\#}$ for finite sets. Therefore we emphasize that the notion *r.e. set of sets* is always meant in the sense of definition 5.6 during the present section. Our general convention is as follows: When a set of sets $\mathcal{W}$ is called r.e., then definition 2.8 applies only if it is explicitly stated by $\mathcal{W}$ is a set of *finite* sets.

Theorem 5.15: The following implications hold:

(i)

$$\mathcal{C} \text{ is canonically enumerable} \Longrightarrow \mathcal{C} \text{ is characteristically enumerable} \Longrightarrow \mathcal{C} \text{ is r.e.}$$

(ii)

$$\mathcal{C} \text{ is completely r.e.} \Longrightarrow \mathcal{C} \text{ is r.e.}$$

Proof: (i) Let $\mathcal{C} = \{ D_x \mid x \in A \}$ with A r.e. Then $\mathcal{C} = \{ C_x \mid x \in f(A) \}$, where f is the function from theorem 5.7, and $f(A)$ is r.e. by theorem 5.10. Similarly, if $\mathcal{C} = \{ C_x \mid x \in A \}$, then $\mathcal{C} = \{ W_x \mid x \in g(A) \}$, where g is again taken from theorem 5.7.

(ii) Trivial. □

Remark 5.2: Of course it would also be possible to define completely characteristically and completely canonically enumerable classes of sets. However, the properties 'canonically enumerable' and 'completely canonically enumerable' coincide, since every finite set has only one canonical index. Moreover, only the empty set of sets would be completely characteristically enumerable, as the following result shows.

Theorem 5.16: Let $\emptyset \neq \mathcal{C} \subseteq \mathcal{R}_*^{(1)}$. Then the set

$$A = \{ x \mid \phi_x \text{ is a characteristic function } \& \ C_x \in \mathcal{C} \}$$

is not r.e.

Proof: The proof parallels the proof in example 5.2 that Tot, Const and Inf are not r.e. Let $B \in \mathcal{C}$, $f = c_B$. Define g by

$$\phi_{g(x)}(y) = \begin{cases} f(y) & \text{if } (\forall z < y)\big(\neg T(z, x, x)\big), \\ \uparrow & \text{otherwise.} \end{cases}$$

If $x \in K$, then $\phi_{g(x)}$ has finite domain, hence $g(x) \notin A$. If $x \notin K$, then $\phi_{g(x)} = f$, hence $g(x) \in A$. Thus $\overline{K} \leq_1 A$ via g, which proves the assertion. □

Example 5.5: (i) Let $A \subseteq \omega$ be any r.e. set. It follows from exercise 5.5 (ii) that the sets

$$\{\, C \mid C \text{ is finite} \,\&\, C \subseteq A \,\}, \text{ and}$$
$$\{\, C \mid C \text{ is finite} \,\&\, C \cap A \neq \emptyset \,\}$$

are canonically enumerable.

(ii) Trivially, $\mathcal{E}^{(1)}$ is completely r.e.

(iii) The set $\{\, C \mid C \text{ is r.e.} \,\&\, C \cap A \neq \emptyset \,\}$ is completely r.e., since the following predicates are r.e.:

$$\begin{array}{ll} \lambda xy.y \in W_x & \text{(theorem 3.15)} \\ \lambda xy.y \in A & \text{(theorem 3.10)} \\ \lambda xy.y \in W_x \,\&\, y \in A & \text{(theorem 5.10)} \\ \lambda x.(\exists y)(y \in W_x \,\&\, y \in A) & \text{(theorem 3.9)} \end{array}$$

The last of these predicates corresponds to the set $\{\, x \mid W_x \cap A \neq \emptyset \,\}$. In this example we pointed out all results used to show that the set considered is r.e.; from now on such results will be used tacitly.

(iv) We show that $\mathcal{PR}_*^{(1)}$ is characteristically enumerable. Recall that $\psi'(x, y)$ is a universal function for $\mathcal{PR}$. Let f be recursive such that for all x, y

$$\phi_{f(x)}(y) = \operatorname{sig}\big(\phi'_x(y)\big) = \operatorname{sig}\big(\psi'(x, y)\big).$$

Let $A = \operatorname{ran}(f)$. Clearly, ϕ_x is a primitive recursive characteristic function for all $x \in A$, hence $\{\, C_x \mid x \in A \,\} \subseteq \mathcal{PR}_*^{(1)}$. Conversely, let $C \in \mathcal{PR}_*^{(1)}$. Then $c_C = \phi'_u$ for some u, and $\phi_{f(u)}(y) = \operatorname{sig}\big(\phi'_u(y)\big) = \phi'_u(y)$ for all y, since $\phi'_u(y) \in \{0, 1\}$. Thus $f(u) \in A$, and $C_{f(u)} = C$.

The class $\mathcal{R}_*^{(1)}$ is of particular interest, so it is dealt with in a theorem:

Theorem 5.17: $\mathcal{R}_*^{(1)}$ is r.e., but not characteristically enumerable.

Proof: We show first that $\mathcal{R}_*^{(1)}$ is not characteristically enumerable. This proof parallels that of theorem 5.1. Assume A is an r.e. set such that ϕ_x is a characteristic function for all $x \in A$, and $\mathcal{R}_*^{(1)} = \{\, C_x \mid x \in A \,\}$. Let f be a recursive function with range A. Define

$$g(x) = 1 \dot{-} \phi_{f(x)}(x).$$

Then g is the characteristic function of a recursive set C, since $\phi_{f(x)}$ is total for all x. Also, $g(x) \neq \phi_{f(x)}(x)$, hence $g \neq \phi_z$ for all $z \in A$. Therefore $C \notin \{\, C_x \mid x \in A \,\}$, contradicting the assumption about A.

To show that $\mathcal{R}_*^{(1)}$ is r.e., we define a recursive function f such that $W_{f(0)}$, $W_{f(1)}$, $\ldots$ is an enumeration of all recursive sets. If ϕ_x is a characteristic function, then $W_{f(x)} = C_x$, but if ϕ_x is not total, then $W_{f(x)}$ will be finite. Intuitively, as long as $\phi_x(y){\downarrow}$ for all y considered so far, all y with $\phi_x(y) = 1$ are enumerated in $W_{f(x)}$, but as soon as for some y, $\phi_x(y){\uparrow}$, then no further elements are enumerated.

Formally, f is defined as follows:

$$\phi_{f(x)}(y) = \begin{cases} \sum_{z<y} \phi_x(z) & \text{if } \phi_x(y) = 1, \\ \uparrow & \text{otherwise.} \end{cases}$$

Clearly, for all x, $W_{f(x)}$ satisfies the properties specified in the preceding paragraph. It follows that $W_{f(x)}$ is recursive for all x, and conversely that for every recursive set $C \subseteq \omega$ there exists an x such that $C = W_{f(x)}$. Thus

$$\mathcal{R}_*^{(1)} = \{ W_x \mid x \in \mathrm{ran}(f) \}.$$

□

We will also show that $\mathcal{R}_*^{(1)}$ is not completely r.e., but only after theorem 5.18.

Exercise 5.9: Find sets $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ of *finite* subsets of ω such that

(i) [4] $\mathcal{C}_1$ is characteristically enumerable, but not canonically enumerable.

(ii) [4] $\mathcal{C}_2$ is recursive enumerable, but not characteristically enumerable.

(iii) [2] $\mathcal{C}_3$ is not r.e.

The following theorem, which is due to Rice, Shapiro, McNaughton and Myhill, provides a criterion for r.e. set index sets.

Theorem 5.18: A set $\mathcal{W} \subseteq \mathcal{E}^{(1)}$ is completely r.e. if and only if there exists a canonically enumerable set $\mathcal{D}$ of finite sets such that

$$\mathcal{W} = \{ C \in \mathcal{E}^{(1)} \mid (\exists D \in \mathcal{D})(D \subseteq C)\}.$$

Proof: Assume first that $\mathcal{W} = \{ W_x \mid (\exists D \in \mathcal{D})(D \subseteq W_x)\}$, where $\mathcal{D}$ is canonically r.e.: $\mathcal{D} = \{ D_x \mid x \in A \}$, A r.e. Then

$$W_x \in \mathcal{W} \iff (\exists y)(y \in A \,\&\, D_y \subseteq W_x);$$

this representation shows that the set $\{ x \mid W_x \in \mathcal{W} \}$ is r.e., using exercise 5.5 and theorem 3.9.

Conversely, assume that $\mathcal{W}$ is completely r.e., that is, the set $A = \{ x \mid W_x \in \mathcal{W} \}$ is r.e. Let

$$\mathcal{D} = \{ C \mid C \in \mathcal{W} \,\&\, C \text{ is finite} \}.$$

If f, g are as in theorem 5.7, then for all x

$$D_x \in \mathcal{D} \iff W_{gf(x)} \in \mathcal{W} \iff gf(x) \in A,$$

hence $\mathcal{D}$ is canonically enumerable by theorem 3.10. It remains to show that $\mathcal{W} = \mathcal{W}'$, where $\mathcal{W}' = \{ C \in \mathcal{E}^{(1)} \mid (\exists D \in \mathcal{D})(D \subseteq C) \}$.

Assume first that $C \in \mathcal{W}'$, but $C \notin \mathcal{W}$. Let $D \in \mathcal{D}$ be such that $D \subseteq C$, and let f be partial recursive with domain C. Define g by

$$\phi_{g(x)}(y) = \begin{cases} f(y) & \text{if } x \in K \vee y \in D, \\ \uparrow & \text{otherwise.} \end{cases}$$

Then

$$W_{g(x)} = \begin{cases} C & \text{if } x \in K, \\ D & \text{if } x \notin K. \end{cases}$$

Since $D \in \mathcal{W}$, but $C \notin \mathcal{W}$, it follows that $\overline{K} \leq_m A$ via g, which leads to the usual contradiction that $\overline{K}$ is r.e. Hence no such C exists, that is, $\mathcal{W}' \subseteq \mathcal{W}$.

Now assume $C \in \mathcal{W}$, but $C \notin \mathcal{W}'$. Then no finite subset of C belongs to $\mathcal{W}$. Again, let f be partial recursive with domain C, and let g be recursive such that for all x, y

$$\phi_{g(x)}(y) = \begin{cases} f(y) & \text{if } (\forall z < y)\big(\neg T(z,x,x)\big), \\ \uparrow & \text{otherwise.} \end{cases}.$$

Then

$$W_{g(x)} = \begin{cases} C & \text{if } x \notin K, \\ \text{a finite subset of } C & \text{if } x \in K. \end{cases}$$

Once again, $\overline{K} \leq_m A$ via g, leading to the same contradiction as before.

Combining, it follows that $\mathcal{W} = \mathcal{W}'$. This completes the proof. □

Corollary 5.19: If $\mathcal{W}$ is completely r.e. and $C \in \mathcal{W}$, then

(i) All r.e. sets $C' \supset C$ belong to $\mathcal{W}$;

(ii) Some finite subset of C belongs to $\mathcal{W}$. □

This corollary is somewhat weaker than theorem 5.18, since it provides only a necessary, not a sufficient condition for a set index set to be completely r.e. In fact, if $\mathcal{W}$ satisfies conditions (i), (ii) of corollary 5.19, then there exists a set $\mathcal{D}$ of finite sets such that $\mathcal{W} = \{ C \in \mathcal{E}^{(1)} \mid (\exists D \in \mathcal{D})(D \subseteq C)\}$, but it is not guaranteed that $\mathcal{D}$ is canonically enumerable. Nevertheless, in many applications corollary 5.19 is sufficient, as the following example illustrates.

Example 5.6: Consider once again the sets considered in and introduced before example 5.2. Of course, theorem 5.18 does not apply to the set K_1, which is not an index set. It is possible to modify this theorem and corollary 5.19 to obtain a criterion for function index sets to be r.e., so that the sets Const, Prim and Rec can be handled, see exercise 5.12. At the moment consider the remaining sets, which are set index sets.

We have $\overline{\text{Empt}} = \{ C \mid (\exists D \in \mathcal{D})(D \subseteq C) \}$, where

$$\mathcal{D} = \{ D \subset \omega \mid D \text{ is finite} \mathbin{\&} D \neq \emptyset \} = \{ D_x \mid x \neq 0 \}.$$

This representation shows that $\overline{\text{Empt}}$ is r.e., though the proof given in example 5.2 is simpler. But corollary 5.19 is very helpful when it comes to showing that certain set index sets are *not* r.e. At the moment denote the set $\{ W_x \mid x \in A \}$ by $\mathcal{W}_A$ ($A \subseteq \omega$). If A is one of the sets Empt, $\overline{\text{Tot}}$, Fin, $\overline{\text{Cof}}$ or Rec, then $\mathcal{W}_A$ does not satisfy condition (i) of corollary 5.19, while if A is one of the sets Tot, Inf or Cof, then $\mathcal{W}_A$ fails to satisfy condition (ii). Finally, if $A = \overline{\text{Rec}}$, then $\mathcal{W}_A$ fails on both accounts. In each case it follows that $\mathcal{W}_A$ is not completely r.e., and that A is not r.e.

Exercise 5.10: [3] Give an example of a set $\mathcal{W} \subset \mathcal{E}^{(1)}$ which is not completely r.e., such that corollary 5.19 cannot be applied. Hence either theorem 5.18 or some other argument has to be used.

Exercise 5.11: [2] Show that if $\mathcal{W} \neq \emptyset$ and $\mathcal{W}$ is completely r.e., then $\mathcal{W}$ contains some nonrecursive sets.

Exercise 5.12: [4] Formulate and prove a criterion for r.e. function $\overline{\text{index}}$ sets, analogous to theorem 5.18. Use it to provide another proof that Const, $\overline{\text{Const}}$, Prim and $\overline{\text{Prim}}$ are not r.e.

5.4. The recursion theorem.

We now turn to a rather surprising theorem, which is a very powerful tool for constructing partial recursive functions or r.e. sets with certain properties. Suppose you are given a program P in some programming language, which computes some partial recursive function $f \in \mathcal{P}^{(1)}$. P is complicated and confusing, and you have no feeling as to what the function f might be. Your task is to return a program P' which has to satisfy just the following property: The function $f' \in \mathcal{P}^{(1)}$ computed by P' must not be identical with f. How would you proceed? Of course, P' can be constructed such that $f'(x) = f(x)+1$ for all x, but then $f' = f$ if f is nowhere defined. Or you might design P' such that $f'(x)$ is sometimes independent of $f(x)$, for instance, $f'(x) = x^2+5x$ for x even and $f'(x) = f(x)+2$ for x odd; but this, too does not guarantee that $f' \neq f$, since $f(x)$ might be x^2+5x for x even and undefined for x odd. It turns out that there exists no algorithm which yields a program P' with the desired property for every given P. In the following theorem the programs are of course replaced by indices of partial recursive functions. For clarity, it is first stated in the simplest form. The proof is short, but rather mysterious.

Theorem 5.20 (fixed point- or recursion theorem): For every recursive function f there exists an index u such that $\phi_u = \phi_{f(u)}$.

Proof: Let g be recursive such that for all x, y

$$\phi_{g(x)}(y) = \psi\big(\psi(x,x), y\big) = \begin{cases} \phi_{\phi_x(x)}(y) & \text{if } \phi_x(x)\downarrow, \\ \uparrow & \text{if } \phi_x(x)\uparrow. \end{cases}$$

Let v be an index such that $\phi_v = fg$. Then

$$\phi_{g(v)} = \phi_{\phi_v(v)} = \phi_{fg(v)} = \phi_{f(g(v))}.$$

Hence the number $u = g(v)$ satisfies the required condition. □

A picture which may clarify this proof somewhat is obtained by arranging the functions $\phi_{\phi_x(y)}$ $(x, y \in \omega)$ in a twodimensional infinte array, with $\phi_{\phi_x(y)}$ being interpreted as $\lambda z.\!\uparrow$ if $\phi_x(y)\uparrow$:

$$\begin{array}{lllllllll}
\phi_{\phi_0(0)} & \phi_{\phi_0(1)} & \cdots & & & & & & \\
\phi_{\phi_1(0)} & \phi_{\phi_1(1)} & \cdots & & & & & & \\
\vdots & \vdots & \ddots & & & & & & \\
\phi_{\phi_w(0)} & \phi_{\phi_w(1)} & \cdots & \phi_{\phi_w(w)} & \cdots & \phi_{\phi_w(v)} & \cdots & \phi_{\phi_w(x)} = \phi_x(x) & \\
\vdots & \vdots & & & \ddots & & & & \\
\phi_{\phi_v(0)} & \phi_{\phi_v(1)} & & \cdots & & \phi_{\phi_v(v)} & \cdots & \phi_{\phi_v(x)} = \phi_{f\phi_w(x)} &
\end{array}$$

Here w is an index such that ϕ_w is the function g from the proof above. By definition of g $\phi_{\phi_w(x)} = \phi_{\phi_x(x)}$, which means that the diagonal in this array equals the w-th row. The function f maps every row t into a row t' in the sense that $\phi_{\phi_{t'}(x)} = \phi_{f\phi_t(x)}$. In particular, it maps row w to row v. If $x = v$, then $\phi_{\phi_v(x)}$ is the element of the diagonal corresponding to the element $\phi_{\phi_w(x)}$ of the w-th row, hence $\phi_{\phi_w(x)} = \phi_{\phi_v(x)} = \phi_{f\phi_w(x)}$, which provides the desired fixed-point property.

This argument is viewed in Owings [18] as a diagonal argument which fails, see also Soare [23], p. 36–37. Indeed, if there were a function f such that $\phi_{f(x)} \neq \phi_x$ for all x, then diagonalization would allow to construct a sequence of functions $(\phi_{\phi_t(x)})_{x\in\omega}$ which differs from each of the rows in the array considered, namely $(\phi_{f\phi_x(x)})_{x\in\omega}$ would differ from $(\phi_{\phi_t(x)})_{x\in\omega}$ at the t-th position. However, by theorems 3.2 and 3.4 there exists an index v such that for all x, $\phi_{f\phi_x(x)} = \phi_{\phi_v(x)}$, which means that $(\phi_{f\phi_x(x)})_{x\in\omega}$ actually agrees with the v-th row of the above array.

Example 5.7: (i) Theorem 5.20 implies immediately that there exists an index u such that $\phi_{u+1} = \phi_u$. Using the enumeration of $\mathcal{P}^{(1)}$ introduced in definition 3.2, it is not difficult to find such u; for instance, u may be such that neither u nor $u+1$ is the code number of some $\mathcal{WL}$-program. But the remarkable thing is that such u would also exist if any other enumeration of $\mathcal{P}^{(1)}$ were taken as a basis, so long as it satisfies theorems 3.2 and 3.4.

(ii) Of course, $\phi_u = \phi_{f(u)}$ implies $W_u = W_{f(u)}$, hence theorem 5.20 remains valid if ϕ is replaced by W throughout. For instance, there exists an element u such that $W_u = \{u\}$. To show this, let f be recursive such that $W_{f(x)} = \{x\}$, for instance,

$$\phi_{f(x)}(y) = \begin{cases} 0 & \text{if } x = y, \\ \uparrow & \text{if } x \neq y \end{cases}$$

for all x, y. Then apply theorem 5.20.

Exercise 5.13: [2] Let f be any recursive function. Show that there exist indices x, y, z, u such that

(i) W_x is finite and $|W_x| > f(x)$;
(ii) $\overline{W_x}$ is finite and $|\overline{W_x}| > f(x)$;
(iii) C_x is finite and $|C_x| > f(x)$;
(iv) $\overline{C_x}$ is finite and $|\overline{C_x}| > f(x)$.

(Once one of the parts is solved, the idea should be clear, and the remaining parts will be quite simple.)

Now we discuss possible generalizations of this theorem. They are obtained simply by analysing the proof of theorem 5.20; no new idea is needed. First, the construction of ϕ_u from f is uniform, so there exists a recursive function h such that for all x, $\phi_{h(x)} = \phi_{\phi_x(h(x))}$. Also, the function h will be one-one, since it is constructed by means of the S-m-n-theorem. Applying this theorem again shows next that if f has some additional arguments (parameters) besides the index x, then the fixed point can be computed effectively from them. Finally, it is possible to compute not only just one fixed point, but countably many, either by using countably many indices of f, or, even more simply, by introducing an additional dummy parameter, as in the following proof. The following generalized version of the recursion theorem combines all these observations.

Theorem 5.21: For every m there exists an one-one, primitive recursive function $h \in \mathcal{P}^{(m+2)}$ such that for all x, $\vec{y}_m$, z

$$\phi_{h(x,\vec{y}_m,z)} = \phi_{\phi_x^{(m+1)}(h(x,\vec{y}_m,z),\vec{y}_m)}. \tag{1}$$

(Hence all $h(x, \vec{y}_m, z)$ ($z \in \omega$) are fixed points of $\lambda t.\phi_x^{(m+1)}(t, \vec{y}_m)$.)

Proof: Let g be as in the proof of theorem 5.20. This function is obtained by corollary 3.5, hence it is one-one. The fixed point from theorem 5.20 is $g(v) = gh'(x)$, where $f = \phi_x$ and $\phi_{h'(x)}(y) = \phi_x(g(y)) = fg(y)$; h' is once again one-one and primitive recursive by corollary 3.5. Hence (1) is satisfied by defining h such that $h(x, \vec{y}_m, z) = gh'(w)$, where $\phi_w = \lambda t.\phi_x^{(m+1)}(t, \vec{y}_m)$. The index w is obtained by yet another application of the S-m-n-theorem: $w = h''(x, \vec{y}_m, z)$, where h'', too, is one-one and primitive recursive. Thus the function $h(x, \vec{y}_m, z) = gh'h''(x, \vec{y}_m, z)$ satisfies the required properties. □

Of course, theorem 5.21, like theorem 5.20, remains valid with ϕ replaced by W at the outer level:

$$W_{h(x,\vec{y}_m,z)} = W_{\phi_x^{(m+1)}(h(x,\vec{y}_m,z),\vec{y}_m)}.$$

Exercise 5.14: [1] Prove that if $A \subseteq \omega$ is a function index set, then $A \not\leq_m \overline{A}$.

Exercise 5.15: [3] Let A be the set of all x such that x is the *smallest* index of ϕ_x:

$$A = \{ x \mid (\forall y)(\phi_x = \phi_y \Longrightarrow y \geq x \}.$$

Prove that A is not r.e.

Exercise 5.16: (i) [4] (Smullyan) Let f_0, $f_1 \in \mathcal{R}^{(2)}$. Show that there exist indices x_0, x_1 such that

$$\phi_{x_0} = \phi_{f_0(x_0,x_1)},$$
$$\phi_{x_1} = \phi_{f_1(x_0,x_1)}.$$

(ii) [2] Generalize this result to n functions $f_0, \ldots, f_{n-1} \in \mathcal{R}^{(n)}$.

The applications of the recursion theorem are numerous, and some of them (considerably more interesting than those considered so far) will occur in the next chapter.

6. Reducibility relations: elementary theory.

In definition 5.1 the two relations $\leq_1$, $\leq_m$ were already introduced, and it was noted that if $A \leq_m B$ holds, then every decision algorithm for membership in B implies a corresponding algorithm for A. This means intuitively that the decision problem for B is at least as difficult as that for A. However, $\leq_1$ and $\leq_m$ are only special types of reducibility relations, they are called the *strong reducibilities*. In this chapter the most general reducibility relation will be defined, and some basic facts about all three reducibility relations will be derived. More interesting results follow in (DCC). Many more reducibility relations, all weaker than $\leq_1$ but stronger than $\leq_T$, have been considered in literature, some of which will be touched briefly in the exercises.

6.1. Relativized computability.

The most general reducibility relation of recursive function theory is called *Turing reducibility* and denoted by $\leq_T$. Intuitively, $A \leq_T B$ asserts the following: It would be possible to decide membership in A effectively, if there were an oracle which answers any question of the form 'is $y \in B$?,' whenever this is desired. This intuition has the following natural formal counterpart: $A \leq_T B$ holds iff c_A is computed by some $\mathcal{AL}(\mathcal{F})$-program, where $\mathcal{F}$ contais c_B and possibly some partial recursive functions and recursive predicates. This will indeed be equivalent to our formal definition of $\leq_T$ (definition 6.2).

More generally, it makes sense to ask for every kind of task whether it can be performed by programs using certain oracles. In this way all basic concepts of recursive function theory can be relativized. As oracles we allow not only sets, but also functions, though later on the relation $\leq_T$ will be primarily applied to sets. In analogy to definitions 1.5–1.7 the definition of the relativized concepts will be based not on $\mathcal{AL}(\mathcal{F})$, but on an extension of $\mathcal{WL}$.

Remark 6.1: The following convention was made in section 2.1 for the semantics of $\mathcal{AL}$-programs, and is also valid for the following definition 6.1: If an assignment of the form $Y := f(X_1, \ldots, X_k)$ has to be executed, and f is undefined for the current values of $X_1, \ldots, X_k$, then the program does not terminate.

Definition 6.1: Let $f: \omega^k \to \omega$ be some partial function, referred to as an *oracle* in what follows.

(i) (N) (Special) $\mathcal{WL}(f)$-programs are defined like ordinary $\mathcal{WL}$-programs, except that one additional type of instruction is allowed, namely $Y := f(X_1, \ldots, X_k)$.

(ii) (E) A partial function $g: \omega^n \to \omega$ is *f-partial recursive* (*f-primitive recursive*) iff g is computed by some (special) $\mathcal{WL}(f)$-program (with respect to some tuple of variables); g is f-recursive, iff g is f-partial recursive and total.

(iii) A set $B \subseteq \omega^n$ is *f- (primitive) recursive*, iff c_B is f- (primitive) recursive; B is *f-recursively enumerable*, iff B is the domain of some f-partial recursive function.

(iv) Now let $A \subseteq \omega^k$. A function g is *A- (primitive/partial) recursive*, iff g is c_A-(primitive/partial) recursive. A set B is *A- (primitive) recursive* (*A-r.e.*), iff B is c_A- (primitive) recursive (c_A-r.e.).

Instead of *f-primitive recursive*, *f-recursive* etc. one also says *primitive recursive relative to f*, *recursive relative to f* etc., and similar for the remaining concepts from parts (ii)–(iv) of this definition.

Notation: Compared with the notation introduced in definition 1.5, after exercise 1.11 and in definition 1.7, concepts relativized with respect to an oracle f or A are indicated by a superscript f or A respectively, or superscript (f, n) or (A, n) instead of (n). For instance, $\mathcal{R}^f$ is the set of all f-recursive functions, and $\mathcal{E}^{(A,n)}$ is the set of all A-r.e. sets $B \subseteq \omega^n$.

Remark 6.2: Definition 6.1 is partially of type (N), since the notion of *f-primitive* recursive function or set is usually not dealt with in textbooks. Also, nontotal functions f are usually not considered as oracles. These restrictions also apply in these lecture notes, except for the current section. The following survey of relativized results will be for the most part complete; this means that we also consider relativizations of theorems which concern primitive recursive functions or sets, as well as nontotal oracles. Nontotal oracles are used for one last time in theorem 6.4 (for technical reasons), but otherwise they are of no interest.

We now describe how the theory of oracle computations parallels that of computations without oracle, as developed in the preceding chapters. In view of part (iv) of definition 6.1 it suffices to consider the case where the oracle is a function f; f will be fixed in the following discussion. Virtually all definitions, results and proofs concerning computability have relativized versions, if f is total; if not, then this is only partially true. The *total relativization* of a definition or theorem is obtained by replacing 'primitive recursive,' 'recursive' etc. by 'f-primitive recursive,' 'f-recursive' etc. throughout. It will be seen, however, that some theorems also have *partial* relativizations, which are even stronger than the total ones and still valid.

Consider first the basic definitions from chapter 1. The notions of algorithm and procedure can be relativized to algorithms and procedures which may use oracle f whenever desired, as it was already described above.

The analogues of definitions 1.2–1.4 are not relativizations in the usual sense, but of course the syntax and semantics of $\mathcal{WL}(f)$ have to be fixed. For the syntax this was done in definition 6.1 (i), and the semantics is also clear, taking remark 6.1 into account. This involves an obvious extension of definiton 1.4 (ii). Finally, the relativizations of definitions 1.5–1.7 were again given in definition 6.1.

If f is assumed to be total, then relativized versions of the results of chapter 1 remain valid, with the same proofs as before. An additional assertion should be included in the relativization of theorem 1.1, so as to deal with the case were P consists of a single assignment statement of the form $Y := f(X_1, \ldots, X_k)$. However, if f is *not* assumed to be total, then the analogue of theorem 1.2 and the inclusions $\mathcal{PR}^f \subseteq \mathcal{R}^f$, $\mathcal{PR}^f_* \subseteq \mathcal{R}^f_*$ in the analogue of theorem 1.3 obviously fail.

In chapter 2 definitions 2.1, 2.8 and 2.15 relativize in a straightforward way, whereas all other definitions remain uneffected by relativization. Many results simply state that certain specific functions and predicates are primitive recursive, and in this case no relativization is needed, of course. (If g is primitive recursive, then a fortiori g is f-primitive recursive.) Almost all other results of this chapter relativize readily, and the proofs remain virtually unchanged. Of course, in the proof of the relativized version of theorem 2.1 one need no longer eliminate assignment statements of the form $Y := f(X_1, \ldots, X_k)$ if f is the oracle function, but other assignments of the form $Y := g(X_1, \ldots, X_n)$ still have to be eliminated.

The only exception concerns the relativization of theorem 2.4 (ii) in the case where $\mathcal{F}$ contains f-primitive recursive, nontotal functions $g_1, g_2, \ldots$. In this case a predicate defined by a Boolean $(\mathcal{V},\mathcal{F})$-expression like $P\big(g_1(x), g_2(x)\big)$ does not always make sense. If it makes sense, as in the case $g_1(x) = g_2(x)$, then it is not necessarily f-recursive. This fact also effects parts (iii) and (iv) of example 2.3, as the reader may check. All these difficulties disappear, however, if the oracle f is total.

Matters become more interesting in chapter 3. Definition 3.2 was based on list representations $\hat{I}$, $\hat{P}$ and $\hat{\mathcal{C}}$ of statements I, programs P and configurations $\mathcal{C}$. This definition extends easily to the relativized case, except that one additional case has to be considered in the definition of $\hat{I}$. Thus we obtain enumerations $(\phi_0^{(f,n)}, \phi_1^{(f,n)}, \ldots)$ of $\mathcal{P}^{(f,n)}$ $(n \in \omega)$, with corresponding universal functions $\psi^{(f,n)}$. Similarly, the definitions of the functions and predicates in theorem 3.1 have to be modified; their relativized versions are f-primitive recursive, and, with the exception of next, even primitive recursive. Thus theorems 3.1 and 3.2 do indeed relativize, using the same proof as before. Moreover, the construction of the program computing $\psi^{(f,n)}$ does not depend on the oracles f, only on the number of its arguments. Therefore we get

Theorem 6.1: For all k, n there exists an index u_0 such that for every oracle $f\colon \omega^k \to \omega$

$$\phi_{u_0}^{(f,n+1)} = \lambda u\vec{x}_n.\phi_u^{(f,n)}(\vec{x}_n).$$

The other two fundamental results of chapter 3, on which most of recursive function theory is based, were theorems 3.3 and 3.4. The analogue of theorem 3.3 will be discussed shortly. For theorem 3.4 we can do even better: In its relativization the analogue of the function s_{mn} is still primitive recursive (not just f-primitive recursive), and independent of the oracle f. The proof proceeds as for the original theorem 3.4; only one additional case is needed in the definition of the function h. The same is true for theorem 3.6; in this case the function ind need not even be modified. Thus we have:

Theorem 6.2 (relativized S-m-n-theorem): For all m, n there exists a one-one primitive recursive function s_{mn}, such that for every oracle f and for all u, $\vec{x}_m$, $\vec{y}_n$

$$\phi^{f,n}_{s_{mn}(u,\vec{x}_m)}(\vec{y}_n) = \phi_u^{f,m+n}(\vec{x}_m, \vec{y}_n).$$

Theorem 6.3 (relativized index theorem): There exists a one-one primitive recursive function ind such that for every oracle f and for all n, u, s

$$\phi^{f,n}_{\mathrm{ind}(u,s)} = \phi_u^{(n)}.$$

Here the analogue of the function s_{mn} from theorem 3.4 was again denoted by s_{mn}. If the function s_{mn} satisfying theorem 6.2 is constructed in a straightforward manner, then it also satisfies theorem 3.4, and we declare with hindsight that this is the final definition of s_{mn}.

As a consequence, many other theorems also have partial relativizations which are stronger than the total ones, namely those which involve a primitive recursive function constructed by theorem 3.4. For instance, corollary 3.5 relativizes as follows: If $g(\vec{x}_m, \vec{y}_n)$ is f-partial recursive, then there exists a one-one, primitive recursive function h such that for all $\vec{x}_m$, $\vec{y}_n$

$$\phi^{(f,n)}_{h(\vec{x}_m)}(\vec{y}_n) = g(\vec{x}_m, \vec{y}_n).$$

Among the remaining definitions of chapter 3, the relativization of 3.4, 3.6 and 3.7 and 3.8 is straightforward, and for the remaining ones no relativization makes sense. The relativization of definition 3.4 leads to the following notation, which parallels $\mathcal{PR}^f$, ϕ_u^f etc.: $W_u^{(f,n)} = \mathrm{dom}(\phi_u^{f,n)})$, and $W_u^f = \mathrm{dom}(\phi_u^f)$. Thus $(W_0^{(f,n)}, W_1^{(f,1)}, \ldots)$ is an enumeration of $\mathcal{E}^{(f,n)}$. The definition of the sets K_0 and K is also relativized:

$$K_0^f = \{\, (x,y) \mid \phi_x^f(y){\downarrow} \,\},$$
$$K^f = \{\, x \mid \phi_x^f(x){\downarrow} \,\}.$$

If f is total, then all remaining theorems of this chapter also relativize, and the proofs remain unchanged. On the other hand, it will be seen in section 6.3 that theorem 3.3 does not necessarily relativize unless f is total, see exercise 6.16 and the remark preceding it. The same is true for other theorems whose proofs use the Kleene predicate. In fact, the only other theorems of chapter 3 which relativize without the condition that f is total are 3.10, 3.14 (i) and 3.15. In some other cases the relativization remains partially correct; for instance the implication $\Longleftarrow$ relativizes in parts (ii) and (iii) of theorem 3.14.

Exercise 6.1: [2] What goes wrong if one tries to relativize the proof of theorem 3.3?

In order to obtain relativized versions of the results of chapter 4 one has to extend the formalisms for defining the partial recursive functions considered in that chapter, so as to allow the use of oracles. This is most easily accomplished for the formalism of μ-recursion: one only has to include the oracle function f in the set of initial functions in definition 4.1. Thus theorem 4.1 generalizes as follows: Let I be defined as in definiton 4.1, $I' = I \cup \{f\}$. Then

(i) A function is f-primitive recursive if and only if it can be defined from functions in I' by (repeated application of) composition and primitive recursion.

(ii) A function is f-partial recursive if and only if it can be defined from functions in I' by (repeated application of) composition, primitive recursion and minimization.

Turing machines with oracles can be defined in various ways. One possibility, most commonly considered in textbooks, is to consider only sets $A \subseteq \omega$ as oracles, and to use a variant of TM's with two tapes which are infinite in only one direction. The second tape is a read-only oracle tape; its x-th cell contains the symbol 1, if $x \in A$, and the symbol 0 if $x \notin A$. The first tape is the working tape which initially contains the input. Such TM's can indeed compute exactly the A-partial recursive functions, decide exactly the A-recursive sets and accept exactly the A-r.e. sets. The restriction of oracles to subsets of ω is not very serious, it is the kind of oracle most commonly considered.

It is also possible to consider Markov algorithms with oracles, but only in a somewhat artificial way; in this case a more substantial modification of definition 4.8 is needed. Using appropriate definitions, the relativized models of computation are indeed equivalent, and the analogues of theorems 4.4–4.6 hold.

Another result from chapter 4, theorem 4.3, can be relativized by considering the notion of f-machine. It is defined as in definiton 4.3, except that $(\mathrm{next}_M)_\omega$ only has to be f-recursive rather than recursive. If f is total, then the following equivalence holds: Let $g\colon A_1 \times \cdots \times A_n \to A$, where $A_1, \ldots, A_n$, A are countable sets for which coding schemes have been defined. Then g is computed by some f-machine M if and only if g is partial recursive.

Consider now the concepts introduced in chapter 5. Definitions 5.4 and 5.5 are not effected by relativization. Relativized versions of the reducibility relations from definition 5.1 are obtained readily, but they are of very little importance in recursive function theory. For definitons 5.2, 5.3 and 5.6 the relativization is again straightforward. In analogy to $W_u^{(f,n)}$ we denote the set A which has f-characteristic index u, that is, which satisfies $c_A = \phi_u^{(f,n)}$, by $C_u^{(f,n)}$, if $\phi_u^{(f,n)}$ is a characteristic function. Concerning definition 5.6, various notions obtained by partial relativization may also be investigated, for instance sets $\mathcal{C} \subseteq \mathcal{P}^{(1)}$ such that $\{ x \mid \phi_x \in \mathcal{C} \}$ is f-r.e. Finally, there are also relativizations of the sets K_1, Empt etc. introduced after theorem 5.5, and they are denoted by applying superscript f, for instance

$$K_1^f = \{ (x, y, z) \mid \phi_x^f(y) = z \},$$
$$\mathrm{Empt}^f = \{ x \mid W_x^f = \emptyset \}.$$

As for the relativizations of the theorems, the situation in chapter 5 is similar as in chapter 3: The total relativizations all hold for total f, and the proofs remain as before. This will also be the case for all results of pure recursive function theory presented from now on. We will not deal any further with the question which relativizations also hold for nontotal f. The crucial point is that the whole theory extends to relativized computations if the oracle is a total function or a set.

To complete the survey of relativized results of chapter 5 for total f, it remains to check which of the theorems have partial relativizations stronger than the total ones. As noted above, functions constructed by the S-m-n-theorem or theorem 3.6 remain primitive recursive upon relativization. This applies to the functions denoted f, g, ... in theorems 5.7, 5.9, 5.10, 5.12 and 5.21. (Of course f has to be renamed, since f now denotes the oracle.) Note also that all these functions are independent of the oracle; this is another consequence of theorems 6.2 and 6.3. Thus we obtain results like the following: There exist primitive recursive functions h, g such that for every oracle f and for all x

(i) $D_x = C^f_{h(x)}$;

(ii) If ϕ_x^f is a characteristic function, then $C_x^f = W^f_{g(x)}$.

Finally, the reducibility relations established in example 5.2 and exercise 5.2 also hold via functions g which were constructed by the S-m-n-theorem. Hence relativization yields $\overline{K^f} \leq_1 \mathrm{Empt}^f$, $\overline{K^f} \leq_1 \overline{\mathrm{Tot}^f}$ etc., without relativizing the relation $\leq_1$.

From now on the reader should have no difficulty finding appropriate relativizations of the definitions and theorems to be presented; they no longer be explicitly stated.

Exercise 6.2: [0] (i) Let f be primitive recursive. Show that a function g is f- (primitive/partial) recursive iff g is (primitive/partial) recursive, and a set A is f- (primitive) recursive or f-r.e. iff A is (primitive) recursive or r.e. respectively.

(ii) Let f be partial recursive. Show that a function g is f- (partial) recursive iff g is (partial) recursive, and a set A is f-recursive or f-r.e. iff A is recursive or r.e. respectively.

Exercise 6.3: Let α, β and γ be elements of $\mathcal{NF} \cup \mathcal{NS}$ (functions or sets). Prove:

(i) [0] α is α-primitive recursive.

(ii) [1] If α is β- (primitive) recursive and β is γ- (primitive) recursive, then α is γ- (primitive) recursive.

Exercise 6.4: Let $f: \omega^n \to \omega$.

(i) [1] Show that if f is total, then $\mathrm{gr}(f)$ is f-primitive recursive and f is $\mathrm{gr}(f)$-recursive.

(ii) [2] Assuming again that f is total, show that f is not necessarily $\mathrm{gr}(f)$-primitive recursive.

(iii) [1] Without the assumption that f is total, show that $\mathrm{gr}(f)$ is f-r.e., but not necessarily f-recursive, and f is $\mathrm{gr}(f)$-partial recursive.

(Compare theorem 3.13.)

Exercise 6.5: [2] Let A be an infinite subset of ω. Recall that p_A denotes the principal function of A (definition 5.5). Show that p_A is A-recursive but not necessarily A-primitive recursive, and A is p_A-primitive recursive. (Compare theorem 5.13).

Exercises 6.2–6.5 dealt with very simple facts, but the following is more substantial:

Exercise 6.6: (i) [3] Show that $\overline{\mathrm{Tot}}$ is K-r.e.

(ii) [3] Show that Cof is Tot-r.e.

In advanced constructions the following kind of phrase is often used: 'The computation of $\phi_u^{(f,n)}(\vec{x}_n)$ terminates after s steps, yields output y, and the oracle value $f(\vec{z}_k)$ is used only for $\vec{v}_k \in A$.' Here A is some finite set. Intuitively, the meaning of this phrase is obvious, nor is it difficult to make it mathematically precise. We already indicated how the notion of computation from definition 1.4 (iv) can be relativized, so the number of steps is also determined. The assertion that the oracle value $f(\vec{v}_k)$ is used only for $\vec{v}_k \in A$ has the following meaning: When the program with code number u and oracle f is executed for input $\vec{x}_n$, then an instruction of the form $W := f(V_1, \ldots, V_n)$ is executed only when $\vec{v}_n \in A$, where v_i is the current value of V_i. It is equivalent to the formalism-independent statement which asserts that the same result is still obtained if the oracle function f were replaced by the function f' defined by

$$f'(\vec{v}_k) = \begin{cases} f(\vec{v}_k) & \text{if } \vec{v}_k \in A, \\ \uparrow & \text{if } \vec{v}_k \notin A. \end{cases}$$

Indeed, if the computation is performed relative to the oracle f' instead of f, then nothing changes, except that by our conventions no result is obtained if one tries to execute an instruction of the form $Y := f'(V_1, \ldots, V_k)$, where the current value of $\vec{V}_k$ is not in A.

Let a finite function $f': \omega^k \to \omega$, an index u, an input vector $\vec{x}_n$ and a number s be given. It is intuitively clear that one can decide effectively whether or not $\phi_u^{(f',n)}(\vec{x}_n)$ terminates in at most s steps, and if so, what the result of this computation is: Just perform s steps of the computation. This is reflected by the following

Theorem 6.4: Let n, k be fixed. There exists a primitive recursive predicate (P) $T_0 = T_0^{k,n}$ on $\omega \times \mathcal{F}(\omega^k, \omega) \times \omega^n \times \omega \times \omega$ such that

(i) For all u, F, $\vec{x}_n$, y

$$(\exists s)\big(T_0(u, F, \vec{x}_n, s, y)\big) \iff \phi_u^{(F,n)}(\vec{x}_n) = y.$$

(ii) If $T_0(u, F, \vec{x}_n, s, y)$, then $s \geq \max\big(\iota_n(\mathrm{dom}(F))\big)$, $s \geq y$ and $T_0(u, F', \vec{x}_n, s', y)$ holds for all $F' \supset F$ and $s' \geq s$.

Proof: Let $T_0(u, F, \vec{x}_n, s, y)$ be the assertion that $s \geq \max\bigl(\iota_n(\mathrm{dom}(F))\bigr)$, $s \geq y$ and the computation of $\phi_u^{(F,n)}(\vec{x}_n)$ yields the result y in at most s steps. It is clear that T_0 satisfies conditions (i) and (ii). We have to show the T_0 is primitive recursive.

Let P be the $\mathcal{WL}(f)$-program from theorem 6.1 which computes $\psi^{(f,n)}$ for any oracle f with respect to U, $X_1, \ldots, X_n$ and Y. A program P' which computes c_{T_0} with respect to U, F, $X_1, \ldots, X_n$, S, Y' and Z is obtained by the following steps 1–5. In the comments we will sometimes refer to a computation of P and a computation of P' which simulates it; in this case the input for P and P' will be denoted by $(u, \vec{x}_n)$ and $(u, F, \vec{x}_n, s, y)$ respectively. The double use of F as a variable and an input value should cause no confusion.

1. Convert P first into a $\mathcal{GL}$-program, then into a program without **goto**-statements, but with a main **while**-loop, as described in part (III) of the proof of theorems 2.1 and 2.2. The resulting program has the form

> **while** $C < l$ **do**
>   **if** $C = 0$ **then** I'_0
>   **else if** $C = 1$ **then** I'_1
>   ⋮
>   **else** I'_{l-1}

2. Attach the following sequence of statements at the end of the program:

if $Y = Y'$ **then** $Z := 1$; END;

This modification guarantees that if P yields output y, and its computation is simulated completely by P', then P' yields output 1, otherwise P' does not change Z and yields output 0. In some cases P' will not terminate the simulation by jumping to the empty statement labelled END.

3. Insert the statement

if $S < \max\bigl(\iota_n(\mathrm{dom}(F))\bigr) \vee S < Y'$ **then goto** END

at the beginning. The use of the expression $\max\bigl(\iota_n(\mathrm{dom}(F))\bigr)$ is justified by exercises 2.18 and 2.20, and similarly for the expressions $(V_1, \ldots, V_k) \in \mathrm{dom}(F)$ and $F(V_1, \ldots, V_k)$ to be introduced in the next step.

4. Within the **then**- and **else**-parts of the conditional statement which forms the body of the **while**-loop, replace every assignment of the form $W := f(V_1, \ldots, V_k)$ by the following conditional statement:

> **if** $(V_1, \ldots, V_k) \in \mathrm{dom}(F)$ **then** $W := F(V_1, \ldots, V_k)$
> **else goto** END **end**

This modification guarantees that P' will terminate the simulation and yield output 0 if the computation of P tries to use an oracle value $F(\vec{v}_k)$ which is not defined.

5. Finally, let I be the body of the **while**-loop as obtained after step 4. Replace it by the following sequence of statements:

$S' := S' + 1$;
if $S' > S$ **then** Goto END

This modification introduces S' as a counter for the number of instructions to be executed by P. If P' needs more than s steps to compute $\phi_u^{(F,n)}(\vec{x}_n)$, then P' does not finish its simulation and yields output 0.

Clearly the **while**-loop of P' is bounded by $S+1$. Thus theorem 2.3 applies, and the proof is complete. □

Note that the characterization of $T_0^{k,n}$ in theorem 6.4 is formalism-independent, though the proof of that theorem is not. $T_0^{k,n}$ plays a similar rôle as the Kleene predicate $T^{(n)}$, though the analogy is not complete. It is clear that the phrase 'the computation of $\phi_u^{(F,n)}(\vec{x}_n)$ terminates (yields a result) in at most s steps' also represents a primitive recursive predicate of u, F, $\vec{x}_n$ and s, using a similar proof as in theorem 6.4.

Exercise 6.7: [4] Show that for every oracle $f\colon \omega^k \to \omega$ and every f-partial recursive function $\phi_u^{(f,n)}$ there exists an f-partial recursive function $\chi_u^{(f,n)}\colon \omega^n \to \mathcal{F}(\omega^n, \omega)$ such that for all $\vec{x}_n$

(a) $\chi_u^{(f,n)}(\vec{x}_n)\downarrow \iff \phi_u^{(f,n)}(\vec{x}_n)\downarrow$.

(b) Assume $\chi_u^{(f,n)}(\vec{x}_n)\downarrow$, $\chi_u^{(f,n)}(\vec{x}_n) = F$. Then $F \subseteq f$ and $\phi_u^{(f,n)}(\vec{x}_n) = \phi_u^{(F,n)}(\vec{x}_n)$. (Thus if a computation relative to an oracle f converges, then there also exists a finite $F \subseteq f$ such that the corresponding computation relative to F also converges and yields the same result.)

6.2. Basic facts about reducibility relations.

Definition 6.1 leads directly to the third reducibility relation to be considered in this chapter:

Definition 6.2: Let α, $\beta \in \mathcal{NF} \cup \mathcal{NS}$ be total functions or sets. Then α is *Turing-reducible* to β ($\alpha \leq_T \beta$), iff α is β-recursive.

This definition will be applied primarily for the case where α and β are subsets of ω. It does not provide anything new compared with definition 6.1, but the emphasis will from now on be a different one: we no longer deal primarily with computations relative to some fixed oracle f, but with the structure of 2^ω under $\leq_T$, as well as $\leq_m$ and $\leq_1$.

Definition 6.3: Let $\leq_r$ be any reducibility relation, A, $B \subseteq \omega$.

(i) *A has smaller r-degree* than B ($A <_r B$), iff $A \leq_r B \,\&\, \neg(B \leq_r A)$.

(ii) *A has the same r-degree* as B ($A \equiv_r B$), iff $A \leq_r B \,\&\, B \leq_r A$.

(iii) The *r-degree* of A, $\deg_r(A)$, is the set $\{\, B \subseteq \omega \mid B \equiv_r A \,\}$.

(iv) If $\mathbf{a}$ und $\mathbf{b}$ are r-degrees, then $\mathbf{a} \leq \mathbf{b}$ iff $A \leq_r B$ for $A \in \mathbf{a}$, $B \in \mathbf{b}$. (This relation is well-defined, see theorem 6.5 (iv).) Also, $\mathbf{a} < \mathbf{b}$ iff $\mathbf{a} \leq \mathbf{b} \,\&\, \mathbf{a} \neq \mathbf{b}$, or, equivalently, $A <_r B$ for $A \in \mathbf{a}$ and $B \in \mathbf{b}$.

(v) An r-degree is recursive (r.e.), iff it contains at least one recursive (r.e.) set.

In the following theorem a number of simple facts about the reducibility relations considered are summarized, including some that have already been shown earlier.

Theorem 6.5: In each of the following assertions let A, B, C be subsets of ω, and let $\leq_r$ be one of the relations $\leq_1$, $\leq_m$, or $\leq_T$.

(i) The following implications hold:

$$A \leq_1 B \Longrightarrow A \leq_m B \Longrightarrow A \leq_T B.$$

(ii) The relations $\leq_r$ are reflexive and transitive, that is, for all A, B, C we have

$$A \leq_r A,$$
$$A \leq_r B \,\&\, B \leq_r C \Longrightarrow A \leq_r C.$$

(iii) The relations $\equiv_r$ are equivalence relations.

(iv) For r-degrees $\mathbf{a}$, $\mathbf{b}$ the relation $\mathbf{a} \leq \mathbf{b}$ is well-defined, and it is a partial ordering on the set of all r-degrees.

(v) Let C be any fixed recursive set. The following assertions are equivalent:
   (a) A is recursive;
   (b) $A \leq_T B$ for every B;
   (c) $A \leq_T C$.

Proof: (i) The implication $A \leq_1 B \Longrightarrow A \leq_m B$ is trivial. If $A \leq_m B$ via f, then A is B-recursive, since $c_A(x) = c_B(f(x))$.

(ii) This has already been established in theorem 5.4 and exercise 6.3.

(iii) The relations $\leq_r$ is reflexive by (ii) and symmetric by definition. If $A \equiv_r B$ and $B \equiv_r C$, then $A \leq_r B \leq_r C$ and $C \leq_r B \leq_r A$, hence $A \equiv_r C$, which shows transitivity.

(iv) We have to show: If $A \equiv_r A'$, $B \equiv_r B'$, then $A \leq_r B \iff A' \leq_r B'$. But this follows immediately from the transitivity of $\leq_r$. The assertion that the relation $\leq$ for r-degrees is reflexive and transitive follows from the corresponding properties of $\leq_r$ for sets. Finally, if two r-degrees $\mathbf{a} = \deg_r(A)$, $\mathbf{b} = \deg_r(B)$ satisfy $\mathbf{a} \leq \mathbf{b} \,\&\, \mathbf{b} \leq \mathbf{a}$, then $A \equiv_r B$, hence $\mathbf{a} = \mathbf{b}$.

(v) The implication (a) $\Longrightarrow$ (b) follows from the fact that every recursive set is trivially B-recursive, and (b) $\Longrightarrow$ (c) is trivial. Also, $C \in \mathcal{R}_*$ implies by exercise 6.2 (ii) that every C-recursive set is recursive; this shows $(c) \Longrightarrow (a)$. □

The following example shows that the relations $\leq_1$, $\leq_m$ and $\leq_T$ are not equivalent.

Example 6.1: (i) Theorem 6.5 (v) shows that $\mathcal{R}_*^{(1)}$ forms a single T-degree, denoted by $\mathbf{0}$; it is called the *recursive T-degree*. (Here in comparison to definition 6.3 (v), the definite article may be used.) Every other T-degree is larger than $\mathbf{0}$.

(ii) Let $A = \{0, 1\}$, $B = \{0\}$. Then $A \leq_m B$ via f, where

$$f(x) = \begin{cases} 0 & \text{for } x \leq 1, \\ 1 & \text{for } x \geq 2. \end{cases}$$

However, the relation $A \leq_1 B$ does not hold, since no one-one function can map the two-element set A into the one-element set B. Next, we have $B \leq_T \emptyset$ by theorem 6.5 (v), but $B \leq_m \emptyset$ does not hold, since $B \leq_m \emptyset$ via f would imply $f(0) \in \emptyset$, which is impossible.

(iii) Trivially we have $\overline{K} \leq_T K$. However, $\overline{K} \not\leq_m K$, otherwise it would follow that $\overline{K}$ is r.e. This example also shows that the analogue of theorem 5.5 (ii) does not hold for Turing reducibility.

Exercise 6.8: (i) [2] Give a complete description of all recursive m-degrees, and how they are related by $\leq$.

(ii) [3] Do the same for recursive 1-degrees.

Exercise 6.9: Besides $\leq_1$, $\leq_m$ and $\leq_T$, the most commonly considered reducibility relation is *truth-table reducibility.* Intuitively, A is truth-table reducible to B ($A \leq_{tt} B$), iff for given x one can effectively compute a tuple $(y_1, \ldots, y_m)$ and a Boolean function τ with the following property: If α is the truth value of $x \in A$, and β_i the truth value of $y_i \in B$ $(1 \leq i \leq m)$, then $\alpha = \tau(\beta_1, \ldots, \beta_m)$. If $A \leq_{tt} B$ holds, then $c_A(x)$ can be determined by a computation using oracle B, but in contrast to the case $A \leq_T B$, one has to determine in advance, which instances of the oracle will be used, and how the final result depends on these queries.

To formalize the definition of $\leq_{tt}$, the Boolean function τ has to be encoded. This is done by specifying the set of all subsets F of $\{y_1, \ldots, y_m\}$ such that $\tau(\beta_1, \ldots, \beta_m) =$ **true**, where β_i is the truth value of $y_i \in F$. In this case it suffices to encode the set $\{y_1, \ldots, y_m\}$ instead of the tuple $(y_1, \ldots, y_m)$. Thus the following formal definition is obtained: $A \leq_{tt} B$ via $f\colon \omega \to \omega^{\#} \times \omega^{\#\#}$ holds iff f is recursive, and upon putting $f(x) = \big(f_1(x), f_2(x)\big)$ we have for all x

$$x \in A \iff B \cap f_1(x) \in f_2(x).$$

(This condition also implies $F \subseteq f_1(x)$ for all $F \in f_2(x)$.)

(i) [1] Show that for all A, $B \subseteq \omega$

$$A \leq_m B \Longrightarrow A \leq_{tt} B \Longrightarrow A \leq_T B.$$

(ii) [3] Show that $\leq_{tt}$ is reflexive and transitive. (What requires some work is the *formal* proof that $\leq_{tt}$ is transitive; intuitively, the matter is very simple.)

(iii) [1] Show that for all A, $A \equiv_{tt} \overline{A}$. Conclude that the relations $\leq_m$ and $\leq_{tt}$ do not coincide.

(iv) [2] Show that all recursive sets form a single tt-degree.

(v) [4] Use a diagonal argument to show that $A \leq_T B$ does not imply $A \leq_{tt} B$.

We now turn to the next important concept, namely **completeness**. In general, if $\leq_r$ is a reducibility relation and $\mathcal{C}$ is some class of sets (or other objects for which $\leq_r$ is defined), then a set A is *$(\leq_r, \mathcal{C})$-complete*, iff A has maximum r-degree among all elements of $\mathcal{C}$. In recursive function theory $\mathcal{C} = \mathcal{E}^{(1)}$ by default, hence we get

Definition 6.4: Let $\leq_r$ be a reducibility relation. A set $A \subseteq \omega$ is r-complete, iff A is r.e. and $B \leq_r A$ for every r.e. set B.

If $\leq_r$ and $\leq_{r'}$ are reducibility relations such that $A \leq_r B$ implies $A \leq_{r'} B$ for all A, B, then it follows immediately from the definition that every r-complete set is r'-complete. In particular, we have the following implications:

$$A \text{ is 1-complete} \Longrightarrow A \text{ is } m\text{-complete} \Longrightarrow A \text{ is } T\text{-complete}.$$

The following facts are also quite trivial.

Theorem 6.6: Let A be r-complete. Then an r.e. set $B \in \mathcal{E}^{(1)}$ is r-complete iff $A \leq_r B$.

Proof: If B is r-complete, then $A \leq_r B$ by definition. If $A \leq_r B$, then every r.e. set C satisfies $C \leq_r A \leq_r B$. □

If $\mathcal{C}$ is arbitrary, then $(\leq_r, \mathcal{C})$-complete sets may not exist. In chapter 10 (DCC) there will be an important case where they do exist, but the proof is nontrivial. However, in the cases currently considered the matter is quite simple:

Theorem 6.7: The set K is 1-complete, hence a fortiori m-complete and T-complete.

Proof: (i) We have to show $W_u \leq_1 K$ for every u. Let u be given. Let f be a one-one recursive function such that for all x, y

$$\phi_{f(x)}(y) = \phi_u(x).$$

If $x \in W_u$, then $\phi_{f(x)}(y)\downarrow$ for all y, hence $f(x) \in K$. On the other hand, if $x \notin W_u$, then $\phi_{f(x)}(y)\uparrow$ for all y, hence $f(x) \notin K$. Thus $W_u \leq_1 K$ via f. □

Example 6.2: (i) In example 5.2 it was shown that $\overline{\text{Empt}}$ is r.e. and $\overline{K} \leq_1 \text{Empt}$, hence $K \leq_1 \overline{\text{Empt}}$ by theorem 5.4 (i). Hence $\overline{\text{Empt}}$ is also 1-complete by theorems 6.6 and 6.7.

(ii) Other examples of r.e., nonrecursive subsets of ω are K_0 (theorem 3.15) and K_1 (introduced after theorem 5.5). Since they belong to $\mathcal{E}^{(2)}$ and $\mathcal{E}^{(3)}$ respectively, definition 6.4 can be applied only after encoding them via ι_2 and ι_3. We show that $\iota_2(K_0)$ and $\iota_3(K_1)$ are indeed 1-complete. For this purpose it suffices to construct one-one recursive functions $f\colon \omega \to \omega^2$ and $g\colon \omega \to \omega^3$ such that for all x

$$x \in K \iff f(x) \in K_0 \iff g(x) \in K_1,$$

for then $K \leq_1 \iota_2(K_0)$ via $\iota_2 f$ and $K \leq_1 \iota_3(K_1)$ via $\iota_3 g$. The definition of f is trivial: We have $x \in K \iff (x, x) \in K_0$, hence $f = \lambda x.(x, x)$ works. In order to construct g, first define a one-one recursive function h such that for all x, y,

$$\phi_{h(x)}(y) = 0 \cdot \phi_x(y) = \begin{cases} 0 & \text{if } \phi_x(y)\downarrow, \\ \uparrow & \text{if } \phi_x(y)\uparrow. \end{cases}$$

Then for all x, $x \in K \iff \big(h(x), x, 0\big) \in K_1$, hence put $g(x) = \big(h(x), x, 0\big)$.

Relativizing theorem 6.7 to oracle A, we see that $K^A = \{\, x \mid \phi_x^A\downarrow \,\}$ is $(\mathcal{E}^{(A,1)}, \leq_1)$-complete, hence $A \leq_1 K^A$, since A is trivially A-r.e. Combining this with the relativization of theorem 3.15, which states that K^A is not A-recursive, we conclude that $A <_r K^A$ for each of the reducibility relations $\leq_r$ considered. Thus the r-degrees of A, K^A, K^{K^A} etc. form an infinite sequence of increasing r-degrees, and their exists no largest r-degree. The set K^A is also denoted by A' and called the *jump* of A. It will be shown in (DCC) that the jump operator is well-defined for r-degrees, too.

Up to now all r.e. sets considered are either recursive or 1-complete. This leads to the following questions: Do the relations $\leq_1$, $\leq_m$, $\leq_T$ coincide for nonrecursive r.e. sets? Is there any r.e. set which is m-complete but not 1-complete? T-complete but not m-complete? Neither recursive nor T-complete? These questions are indeed non-trvial. Answers will be provided partially in section 6.4, partially in (DCC).

Joins. Does there exist an upper bound for any two r-degrees $\mathbf{a}$, $\mathbf{b}$ ($r = 1$, m or T)? In contrast to many others questions about the structure of r-degrees under $\leq$, this one is quite elementary and will be answered in the next theorem. Given two sets A, B, we wish to construct a set D such that A and B are both r-reducible to D. The idea is to encode the information about A in the even numbers, that about B in the odd ones.

Definition 6.5: Let A, $B \subseteq \omega$. The *join* of A and B, $A \oplus B$, is the set

$$\{\, 2x \mid x \in A \,\} \cup \{\, 2x+1 \mid x \in B \,\}.$$

Theorem 6.8: (i) For all A, $B \subseteq \omega$ $A \leq_1 A \oplus B$ and $B \leq_1 A \oplus B$, hence $\deg_r(A \oplus B)$ is an upper bound of $\deg_r(A)$ and $\deg_r(B)$ if $\leq_r$ is one of the reducibility relations $\leq_1$, $\leq_m$ or $\leq_T$.

(ii) $\deg_m(A \oplus B)$ is the *least* upper bound of $\deg_m(A)$ and $\deg_m(B)$.

(iii) $\deg_T(A \oplus B)$ is the *least* upper bound of $\deg_T(A)$ und $\deg_T(B)$.

Proof: (i) We have $A \leq_1 A \oplus B$ via $\lambda x.2x$, $B \leq_1 A \oplus B$ via $\lambda x.2x+1$.

(ii) It remains to show: If $A \leq_m C$, $B \leq_m C$, then $A \oplus B \leq_m C$. Let $A \leq_m C$ via f, $B \leq_m C$ via g. Then $A \oplus B \leq_m C$ via h, where

$$h(x) = \begin{cases} f(x/2) & \text{for } x \text{ even,} \\ g(\lfloor x/2 \rfloor) & \text{for } x \text{ odd.} \end{cases}$$

Indeed, if x is even, then $x \in A \oplus B \iff x/2 \in A \iff f(x/2) \in C$, and if x is odd, then similarly $x \in A \oplus B \iff \lfloor x/2 \rfloor \in B \iff g(\lfloor x/2 \rfloor) \in C$.

(iii) Let $A \leq_T C$, $B \leq_T C$. Then c_A and c_B are C-recursive, and the same follows for $A \oplus B$, since

$$c_{A \oplus B}(x) = \begin{cases} c_A(x/2) & \text{for } x \text{ even,} \\ c_B(\lfloor x/2 \rfloor) & \text{for } x \text{ odd.} \end{cases}$$

Thus $A \oplus B \leq_T C$. □

Parts (ii) and (iii) show that the m-degrees and the T-degrees each form an upper semilattice. The function h constructed in the proof of (ii) is not necessarily one-one, hence we cannot conclude that $\deg_1(A \oplus B)$ is the least upper bound of $\deg_1(A)$ and $\deg_1(B)$. In fact, it will be shown in exercise 6.21 that the least upper bound of two 1-degrees does not always exist.

Exercise 6.10: [0] Show that $A \oplus B$ is (primitive) recursive (r.e.) iff A and B are both (primitive) recursive (r.e.).

Exercise 6.11: [2] Show that for every countable set of sets $A_i \subseteq \omega$ ($i \in \omega$) there exists a set A such that $A_i \leq_1 A$ holds for all i. Hence every countable set of r-degrees has an upper bound, where $\leq_r$ is one of the reducibility relations considered.

6.3. Recursive isomorphism

In various areas of mathematics certain classes of objects and their properties are studied, and the relevant properties are those which are invariant under certain transformations (often called isomorphisms). Examples are the group isomorphisms in group theory, the homeomorphisms in topology etc. In the theory of recursive functions, this rôle is played by the *recursive isomorphisms*, and the most fundamental concepts are those which are invariant under them.

Definition 6.6: (i) A *recursive permutation* (*recursive isomorphism*) is a bijective recursive function $f\colon \omega \to \omega$. The set of all recursive permutations is denoted by $\mathcal{G}$.

(ii) Two sets A, B are (*recursively*) *isomorphic* ($A \sim B$) via f, iff $f \in \mathcal{G}$ and $f(A) = B$.

(iii) A property P of subsets of ω is *recursively invariant*, iff the following holds: Whenever $A \subseteq \omega$ satisfies P, then so does every $B \sim A$. A set $\mathcal{F} \subseteq \mathcal{NS}^{(1)}$ is recursively invariant if the property of belonging to $\mathcal{F}$ is recursively invariant.

Theorem 6.9: (i) $\mathcal{G}$ is a group with composition as the binary operation.

(ii) $\sim$ is an equivalence relation.

Proof: (i) If f, $g \in \mathcal{G}$, then fg is both recursive and bijective, hence $fg \in \mathcal{G}$. The fact that composition of functions is an associative operation is elementary. The identity function $\lambda x.x$ works as the neutral element. It is also an elementary fact that the inverse of every bijective function is bijective. Thus it remains to show that the inverse of a recursive permutation f is again recursive. This follows from the representation

$$f^{-1}(x) = (\mu y)[f(y) = x].$$

(ii) For every $A \subseteq \omega$ we have $A \sim A$ via $\lambda x.x$. If $A \sim B$ via f, then $B \sim A$ via f^{-1}. Finally, if $A \sim B$ via f and $B \sim C$ via g, then $A \sim C$ via gf. □

Definition 6.7: The equivalence classes of $\sim$ are called *isomorphism classes* (of $\mathcal{NS}^{(1)}$).

In the proof of theorem 6.9 (i), minimization was used in the representation of f^{-1}. Hence it is not obvious, whether or not the primitive recursive permutations also form a group. The next exercise shows that this is not the case.

Exercise 6.12: [4] Show that there exists a bijective primitive recursive function $f\colon \omega \to \omega$ such that f^{-1} is not primitive recursive. (If the right previous exercise is applied, then the solution will be quite short, but still slightly tricky.)

Example 6.3: Clearly, if $A \sim B$, then $A \equiv_1 B$. Hence theorem 5.5 implies that $\mathcal{R}_*^{(1)}$ and $\mathcal{E}^{(1)}$ are recursively invariant. Also, every r-degree is recursively invariant if $\leq_r$ is one of the reducibility relations considered, and so is the set of all r-complete sets.

Exercise 6.13: (i) [2] Show that if A, $B \subseteq \omega$ are recursive sets such that $|A| = |B|$ and $|\overline{A}| = |\overline{B}|$, then $A \sim B$. (Perhaps you have already constructed a function f such that $A \sim B$ via f in the solution of exercise 6.8.)

(ii) Conclude that $\mathcal{PR}_*^{(1)}$ is not recursively invariant. (This indicates that from the point of view of pure recursive function theory the property of being primitive recursive is not as fundamental as that of being (partial) recursive.)

The reason why recursive isomorphism is dealt with in the present chapter is its close relation to the strong reducibility relations. It was already noted that $A \sim B$ implies $A \equiv_1 B$. Surprisingly, the converse of this statement also holds:

Theorem 6.10: (Myhill) For A, $B \subseteq \omega$, $A \sim B$ holds iff $A \equiv_1 B$.

Proof: We have to show the direction $\Longleftarrow$. Let $A \leq_1 B$ via f, $B \leq_1 A$ via g. A function h such that $A \sim B$ via h is constructed in stages. After stage i a finite portion $h_i \subset h$ will be defined, and h will be $\bigcup_{i \geq 0} h_i$. At stage $2n+1$ the value $h(n)$ is defined, which means that $\mathrm{gr}(h_{2n+1}) = \mathrm{gr}(h_{2n}) \cup (n, y)$ for an appropriate y, and at stage $2n+2$, $h(x)$ is defined to be n for an appropriate element x, that is, $\mathrm{gr}(h_{2n+2}) = \mathrm{gr}(h_{2n+1}) \cup (x, n)$. In this way it is ultimately guaranted that $\mathrm{dom}(h) = \mathrm{ran}(h) = \omega$. This kind of proof is called a *forward-backward argument.* Each h_i is one-one and satisfies the condition

$$\big(\forall x \in \mathrm{dom}(h_i)\big)(x \in A \iff h_i(x) \in B). \tag{1}$$

Stage 0. Let h_0 be the nowhere defined function.

Stage $2n+1$. If $h_{2n}(n)\!\downarrow$, then go to stage $2n+2$, that is, $h_{2n+1} = h_{2n}$. Otherwise put $x_0 = n$, $y_0 = f(n)$. If $y_0 \notin \mathrm{ran}(h_{2n})$, let $h^{(2n)}(n) = y_0$. This preserves the condition (1), since $n \in A \iff f(n) \in B$. If on the other hand $y_0 \in \mathrm{ran}(h_{2n})$, then define $x_1 = h_{2n}^{-1}(y_0)$, $y_1 = f(x_1)$. If $y_1 \in \mathrm{ran}(h_{2n})$ then define $x_2 = h_{2n}^{-1}(y_1)$, $y_2 = f(x_2)$ etc. Continue in this fashion until for some k, $y_k \notin \mathrm{ran}(h_{2n})$. Then put $h_{2n}(n) = y_k$, and $h_{2n+1}(x) = h_{2n}(x)$ for $x \neq n$.

To show that this procedure works, the following facts have to be shown:

(α) All x_i, and also all y_i, are different. We show by induction on i that x_i is different from $x_0, \ldots, x_{i-1}$, and y_i is different from $y_0, \ldots, y_{i-1}$. For $i = 0$, nothing has to be shown. Assume by induction hypothesis that $x_0, \ldots, x_{i-1}$, and also $y_0, \ldots, y_{i-1}$ are pairwise different ($i \geq 1$). Now $h_{2n}(x_i) = y_{i-1}$, but $h_{2n}(x_0)\!\uparrow$, and $h_{2n}(x_j) = y_{j-1} \neq y_{i-1}$ for $1 \leq j < i$. Hence $x_i \neq x_j$ for all $j < i$. Also, $y_i \neq y_j$ for all $j < i$, since $y_j = h_{2n}(x_j)$ for all j, and h_{2n} is one-one.

(β) There exists a k such that $y_k \notin \mathrm{ran}(h^{(2n-1)})$, that is, stage $2n+1$ terminates. This follows simply from (α) and the fact the $\mathrm{ran}(h_{2n})$ is finite.

(γ) $x_0 \in A$ if and only if $y_k \in B$. Since $y_i = f(x_i)$, $A \leq_1 B$ via f, $y_i = h_{2n}(x_{i+1})$, and (1) holds for h_{2n}, we have

$$x_0 \in A \iff y_0 \in B \iff x_1 \in A \iff \ldots \iff x_k \in A \iff y_k \in B.$$

It now follows that h_{2n+1} satisfies all required properties.

Stage $2n+2$. If $n \in \mathrm{ran}(h_{2n+1})$, then $h_{2n+2} = h_{2n+1}$. Otherwise an element $\overline{x} \notin \mathrm{dom}(h_{2n+1})$ is determined such that $x \in A \iff n \in B$, and $h_{2n+1}(\overline{x}) = n$, $h_{2n+1}(x) = h_{2n}(x)$ for $x \neq \overline{x}$. The algorithm parallels that for stage $2n+1$, using g instead of f and h_{2n+1}^{-1} in place of h_{2n}.

Finally, $h = \bigcup_i h_i$, which means that $\mathrm{gr}(h) = \bigcup_i \mathrm{gr}(h_i)$. This completes the construction.

The set $\bigcup_i \mathrm{gr}(h_i)$ is indeed the graph of a function, for if it contains two different pairs (x, y) and (x, y'), then there would be some n such that $\mathrm{gr}(h_n)$ already contains both pairs, contradicting the fact that h_n is a function. For the same reason h is one-one. Also, h is onto and total, since $x \in \mathrm{dom}(h_{2x+1})$ and $y \in \mathrm{ran}(h_{2y+2})$ for all x, y. Finally, $A \sim B$ via h, because condition (1) holds for all h_n and therefore also for h.

In the argument just presented the computation of h is described informally. As noted in section 3.4, we wish to indicate how an application of ChT can be avoided in proofs like the current one, hence we provide an $\mathcal{AL}$-program which generates h with respect to H. Every execution of the **for**-loop corresponds to one stage of the above construction. Stage 0 need not be performed by the program, since H is automatically initialized to $\emptyset$. The program uses all parts of exercise 2.20 except (v) and (vi).

```
H: P^(1);
for I := 1 to ∞ do
    begin
        N := ⌊(I + 1)/2⌋;
        if I mod 2 = 1 & N ∉ dom(H) then
            begin
                Y := f(N);
                while Y ∈ ran(H) do
                    Y := f(H^-1(Y))
                def(H, N, Y)
            end
        else If I mod 2 = 0 & N ∉ ran(H) then
            begin
                X := g(N);
                while X ∈ dom(H) do
                    X := g(H(Y))
                def(H, X, N)
            end
    end
```

□

Another application of recursive permutations concerns *acceptable Gödel numberings*. We only consider enumerations of $\mathcal{P}^{(1)}$, but analogous results hold for enumerations of $\mathcal{P}^{(n)}$, $n \geq 2$. Recall once again that theorems 3.2–3.4 are the crucial properties of our standard enumeration ϕ_0, ϕ_1, ..., from which everything else can be derived. The question is: if these theorems are regarded as axioms, are they independent? Or can one of them be derived from the others? It will be seen that every enumeration of $\mathcal{P}^{(1)}$ which satisfies the analogues of theorems 3.2 and corollary 3.5 (in the latter case even without the assumption that f is one-one and primitive recursive) also satisfies the remaining conditions, except that the analogue of s_{mn} may be only recursive rather than primitive recursive. This is the result of exercise 6.15.

Definition 6.8: (See Rogers [21], exercise 2.10, p. 41). The numbering χ_0, χ_1, ... is an *acceptable Gödel numbering*, iff there exist recursive functions f, g such that for all x $\chi_x = \phi_{f(x)}$ and $\phi_x = \chi_{g(x)}$.

Theorem 6.11: Let $\overline{\chi} = (\chi_0, \chi_1 \ldots)$ be an enumeration of $\mathcal{P}^{(1)}$. The following conditions are equivalent:

(i) $\overline{\chi}$ is acceptable.

(ii) There exists a recursive permutation h such that for all x, $\chi_x = \phi_{h(x)}$.

(iii) The function $\lambda ux.\chi_u(x)$ is partial recursive, and for every partial recursive function $k \in \mathcal{P}^{(2)}$ there exists a recursive function s such that for all x, y

$$\chi_{s(x)}(y) = k(x, y).$$

(The function s serves a similar purpose as the functions s_{mn}, but we are not given that it is one-one.)

Proof: (i) $\Longrightarrow$ (ii): Let f, g be given such that definiton 6.8 is satisfied. Just as in the proof of theorem 6.10 the function h is constructed in stages, and it is guaranteed at stages $2n+1$ and $2n+2$ that $n \in \operatorname{dom} h$ and $n \in \operatorname{ran} h$ respectively. Again h_i denotes the portion of h constructed after stage i. The function h_i is one-one and satisfies

$$\big(\forall x \in \operatorname{dom}(h_i)\big)\big(\phi_x = \chi_{h(x)}\big).$$

Stage 0. Let h_0 be the nowhere defined function.

Stage $2n+1$. If $n \in \operatorname{dom}(h_{2n})$, then go to stage $2n+2$. Otherwise define $h_{2n+1}(n) = \operatorname{ind}\big(f(n), t\big)$, where t is chosen minimal such that $\operatorname{ind}\big(f(n), t\big) \notin \operatorname{ran}(h_{2n})$. This works, since $\chi_n = \phi_{f(n)} = \phi_{\operatorname{ind}(f(n),t)}$.

Stage $2n+2$. If $n \in \operatorname{ran}(h_{2n+1})$, then go to step $2n+3$. Otherwise we have to define $h_{2n+2}(x) = n$ for an element x such that $\chi_x = \phi_n$ and $x \notin \operatorname{dom}(h_{2n+1})$. We choose $x = g(y)$, where $\phi_y = \phi_n$ and $g(y) \notin \operatorname{dom}(h_{2n+1})$. The trick is to find such y. By theorem 5.21 we can find effectively for every z an index u_z such that

$$\phi_{u_z} = \begin{cases} \lambda w.z & \text{if } g(u_z) \in \operatorname{dom}(h_{2n+1}), \\ \phi_n & \text{otherwise.} \end{cases}$$

Note that the righthand side of this equation is of the form $\phi_{r(z,u_z)}$ with r recursive. If $g(u_z) \in \operatorname{dom}(h_{2n+1})$, then $\lambda w.z = \phi_{u_z} = \chi_{g(u_z)}$ is one of the functions χ_v with $v \in \operatorname{dom}(h_{2n+1})$, and this can only be the case for finitely many z. Thus let z_0 be minimal such that $g(u_{z_0}) \notin \operatorname{dom}(h_{2n+1})$, and put $y = u_{z_0}$. Then $\phi_y = \phi_{y_n}$, and y satisfies the required conditions.

(ii) $\Longrightarrow$ (iii): Assume that (ii) holds. Then $\lambda ux.\chi_u(x) = \lambda ux.\phi_{h(u)}(x)$ is partial recursive. Also, by corollary 3.5 there exists a recursive function s' such that for all x, y

$$k(x,y) = \phi_{s'(x)}(y) = \chi_{h^{-1}s'(x)}(y).$$

Hence the function $s = h^{-1}s'$ satisfies the required condition.

(iii) $\Longrightarrow$ (i): Assume (iii). A function f such that for all x, $\chi_x = \phi_{f(x)}$ is obtained simply by applying corollary 3.5 to $\lambda xy.\chi_x(y)$. Next let $k = \psi = \lambda xy.\phi_x(y)$, and let s be recursive such that for all x, y $\chi_{s(x)}(y) = k(x,y)$. Then for all x, $\chi_{s(x)} = \lambda y.k(x,y) = \phi_x$, hence the function $g = s$ satisfies the second condition of definition 6.8. □

Exercise 6.14: [4] Write a $\mathcal{AL}(\mathcal{F})$-program which generates the function h constructed in the proof of the implication (i) $\Longrightarrow$ (ii) of theorem 6.11. $\mathcal{F}$ may contain any recursive function whose existence is guaranteed by the recursion- or some other theorem.

Exercise 6.15: [2] Prove that every acceptable Gödel numbering of $\mathcal{P}^{(1)}$ satisfies the analogues of theorems 3.3, 3.4 and 3.6. The analogue of theorem 3.4 only has to be shown for $m = n = 1$; use the standard enumeration $\phi_0^{(2)}$, $\phi_1^{(2)}$, ... of $\mathcal{P}^{(2)}$.

If χ_0, χ_1, ... is an enumeration of $\mathcal{C}^{(1)}$, where $\mathcal{C}$ is some class of functions other than $\mathcal{P}$, then the analogues of theorems 3.2 and 3.4 no longer necessarily guarantee those of theorem 3.3, even if we assume that $\mathcal{C}$ contains $\mathcal{P}$ and satisfies all the closure properties shown for $\mathcal{P}$ in chapter 2. Specifically, let $\mathcal{C} = \mathcal{P}^f$, for an oracle f which will be specified shortly. It was already noted in the survey of relativized results in section 6.1 that the enumeration ϕ_0^f, ϕ_1^f, ... satisfies the analogue of theorems 3.2, 3.4 and 3.6. If the analogue of theorem 3.3 also holds, then the proof of theorem 3.9 carries over, hence its analogue remains valid. But this is not the case for every f, as the following exercise shows.

Exercise 6.16: (i) [2] Let $A \subseteq \omega$, and define

$$f(x) = \begin{cases} 0 & \text{if } x \in A, \\ \uparrow & \text{if } x \notin A. \end{cases}$$

Show that a function g is f-recursive only if g is recursive.

(ii) [1] Now make the additional assumption that A is not r.e. Show that the f-r.e. set $\operatorname{dom} f$ is not the range of any recursive function.

6.4. Creative sets and simple sets.

The material of this section is all contained in Post [20], one of the first papers which provided nontrivial constructions of r.e. sets with certain properties.

It was noted after the proof of theorem 3.15 that $\overline{K}$ is obtained if one applies straightforward diagonalization to construct a set which is not r.e. For every x we have $x \in \overline{K} \iff x \in W_x$, hence x 'witnesses' that $\overline{K} \neq W_x$.

In general a set A is called *productive*, iff A is not r.e., and a witness for $A \neq W_x$ can be found effectively from the index x. Actually, this conditon characterizes *completely productive* sets, whereas productiveness is formally a weaker condition. However, it will be seen later that these two properties are in truth equivalent.

Definition 6.9: (i) A set $A \subseteq \omega$ is *productive*, iff there exists a recursive function f such that for all x

$$W_x \subseteq A \Longrightarrow f(x) \in A - W_x.$$

Such a function f is called a *productive function* for A.

(ii) A set $A \subseteq \omega$ is *completely productive*, iff there exists a recursive function f such that for all x

$$f(x) \in A \iff f(x) \notin W_x.$$

Such a function f is called a *completely productive function* for A.

(iii) A set $A \subseteq \omega$ is (*completely*) *creative*, iff A r.e. and $\overline{A}$ is (completely) productive.

Example 6.4: The discussion above shows that K is creative, and $\lambda x.x$ is a completely productive function for $\overline{K}$.

Theorem 6.12: If $A \leq_m B$ and A is (completely) productive, then B is (completely) productive.

Proof: Let $A \leq_m B$ via g, and let h be a (completely) productive function for A; we have to construct a (completely) productive function f for B. Let f_1 be a recursive function such that for all z, $W_{f_1(z)} = g^{-1}(W_z)$. The existence of such f_1 follows from theorem 5.10: if $g = \phi_u$, and h' is the function h from that theorem, then the function $f_1 = \lambda z.h'(z,u)$ satisfies the required condition. Now let x be given. If h is a completely productive function for A, then $hf_1(x) \in A = g^{-1}(B) \iff hf_1(x) \notin W_{f_1(x)} = g^{-1}(W_x)$, hence $ghf_1(x) \in B \iff ghf_1(x) \notin W_x$. If h is just a productive function for A and $W_x \subseteq B$, then $W_{f_1(x)} \subseteq A$ and $hf_1(x) \in A - W_{f_1(x)}$, hence $ghf_1(x) \in B - W_x$. Hence $f = ghf_1$ is the desired (completely) productive function for B. □

Now since $\overline{K}$ is productive and K is creative, we obtain the following implications:

$$\overline{K} \leq_1 A \Longrightarrow \overline{K} \leq_m A \Longrightarrow A \text{ is completely productive} \Longrightarrow A \text{ is productive,}$$

$$\begin{aligned} &A \text{ is 1-complete} \Longrightarrow A \text{ ist } m\text{-complete} \Longrightarrow A \text{ is r.e. \& } K \leq_m A \\ &\qquad\qquad\qquad\qquad \Longrightarrow A \text{ is completely creative} \Longrightarrow A \text{ is creative.} \end{aligned}$$

We will prove shortly that the converses of these implications also hold, but this requires one more preparation.

Theorem 6.13: If A is productive, then there exists a *one-one* productive function for A.

Proof: Let f be an arbitrary productive function for A. We construct a one-one productive function g; first we describe how $g(x)$ is computed, assuming that $g(0)$, ..., $g(x-1)$ have already been determined. Compute indices u_0, u_1, ... and numbers y_0, y_1, ... such that

$$\begin{aligned} u_0 &= x, \\ y_i &= f(u_i) \text{ für alle } i, \\ W_{u_{i+1}} &= W_{u_i} \cup \{y_i\}. \end{aligned}$$

These numbers can be computed effectively; u_{i+1} is obtained from u_i and y_i using theorems 5.7 and 5.10. Continue computing these elements until for some k (possibly 0) one of the following two cases holds:

(a) $y_k \notin \{g(0), \ldots, g(x-1)\}$. In this case put $g(x) = y_k$.

(b) $y_k \in \{y_0, \ldots, y_{k-1}\}$. In this case put $g(x) = (\mu y)[y \notin \{g(0), \ldots, g(x-1)\}]$.

This computation of $g(x)$ terminates, for if case (b) does not occur, then all y_i are different, which implies that (a) must happen after finitely many steps. If $W_x \not\subseteq A$, then $g(x)$ only has to satisfy the condition $g(x) \notin \{g(0), \ldots, g(x-1)\}$, in order that g will ultimately be one-one; this is guaranteed both in case (a) and in case (b). Now assume $W_x \subseteq A$. Induction on i shows that the following conditions are satisfied for all i:

(i) $W_{u_i} = W_x \cup \{y_0, \ldots, y_{i-1}\} \subseteq A$,

(ii) $y_i \in A - W_{u_i}$.

Condition (i) is satisfied for $i = 0$ due to the assumption $W_x = W_{u_0} \subseteq A$. For every (i) the condition (i) implies (ii), since f is a productive function for A. If (i) and (ii) are satisfied for some (i), then the condition (i) for $i+1$ in place of (i) follows immediately from the definition of $W_{u_{i+1}}$.

It follows now that all y_i are different, hence case (a) rather than (b) occurs. Then $g(x) = y_k$ satisfies the condition $g(x) \in A - W_{u_k} \subseteq A - W_x$, and in addition $g(x)$ is once again different from $g(0)$, ..., $g(x-1)$. Thus g is indeed a one-one productive function for A.

It remains to provide a rigorous proof that g is recursive. This is left as an exercise, as for theorem 6.11. □

Exercise 6.17: [4] Redo exercise 6.14, except that this time your $\mathcal{E}(\mathcal{F})$-program has to generate the function g from the proof of theorem 6.13.

Theorem 6.14: The following equivalences are valid:

$$\overline{K} \leq_1 A \iff \overline{K} \leq_m A \iff A \text{ is completely productive} \iff A \text{ is productive},$$

$$A \text{ is 1-complete} \iff A \text{ ist } m\text{-complete} \iff A \text{ is r.e. \& } K \leq_m A$$
$$\iff A \text{ is completely creative} \iff A \text{ is creative.}$$

Proof: It only remains to show:

$$A \text{ is productive} \Longrightarrow \overline{K} \leq_1 A.$$

Let f be a one-one productive function for A, and let $g(y,x)$ be one-one, recursive, such that for all x, y

$$W_{g(y,x)} = \begin{cases} \{f(y)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \notin K. \end{cases}$$

For instance, corollary 3.5 may be used to define g such that for all x, y, z

$$\phi_{g(y,x)}(z) = \begin{cases} \phi_x(x) & \text{if } z = f(y), \\ \uparrow & \text{falls } z \neq f(y). \end{cases}$$

By theorem 5.21 there exists a one-one recursive function h such that for all x

$$W_{h(x)} = W_{g(h(x),x)} = \begin{cases} \{fh(x)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \notin K. \end{cases}$$

If $x \in K$, then this condition is consistent with f being a productive function for A only if $fh(x) \notin A$. If $x \notin K$, then $W_{h(x)} = \emptyset$ implies that $fh(x) \in A$. Thus $\overline{K} \leq_m A$ via fh. Since moreover f and h are one-one, we even have $\overline{K} \leq_1 A$. □

Exercise 6.18: [4] Let $A \subseteq \omega$, and assume that there exists a partial recursive function f such that for all x

$$W_x \subseteq A \Longrightarrow f(x)\downarrow \,\&\, f(x) \in A - W_x.$$

This condition is weaker than that of definition 6.9 (i) in so far as $f(x)$ need not converge if $W_x \not\subseteq A$. Prove nevertheless that that A is productive. (*Hint:* Use theorem 5.21 again!)

Exercise 6.19: Let A, $B \subseteq \omega$ be r.e. sets.

(i) [0] Show that if A or B is creative, then $A \oplus B$ is creative.

(ii) [4] Show that the converse also holds: $A \oplus B$ is creative only if either A or B is creative. Conclude that $\deg_1(K)$ is not the least upper bound of any two lower 1-degrees. (The latter assertion also follows from Lachlan [12], who proved a considerably stronger theorem.)

Productive sets satisfy yet another important property:

Theorem 6.15: If A is productive, then A contains an infinite r.e. subset.

Proof: Let f be a one-one productive function for A, and let u be an index such that $W_u = \emptyset$. Then for all x, $W_{\mathrm{ind}(u,s)} = \emptyset$, hence $f\big(\mathrm{ind}(u,x)\big) \in A$. Thus if $g(x) = f\big(\mathrm{ind}(u,x)\big)$, then $\mathrm{ran}\, g$ is an infinite r.e. subset of A. □

This theorem provides a first suggestion as to how one might construct an r.e. set A which is neither recursive nor 1-complete: Since by theorem 6.14 A is 1-complete iff $\overline{A}$ is productive, it suffices to guarantee that A is nonrecursive and $\overline{A}$ contains no infinite r.e. set.

Definition 6.10: (i) A set $A \subseteq \omega$ is *immune*, iff A is infinite and contains no infinite r.e. subset.

(ii) $A \subseteq \omega$ is *simple*, iff A is r.e. and $\overline{A}$ is immune.

Theorem 6.15 asserts that no productive set is immune, hence no creative set is simple. Also, it follows immediately from definition 6.10 that an immune set cannot be r.e., hence a simple set cannot be recursive. Hence the set S from the next theorem is neither recursive nor 1-complete (or, equivalently, m-complete). This is our first result which implies the existence of an r.e. set with this property.

Theorem 6.16 (Post [20]): There exists a simple set S.

Proof: The idea is to enumerate all W_x simultaneously for all x; if some $y \geq 2x$ appears in the enumeration of W_x, then add this y into S and terminate the enumeration of W_x. In this case we say that *y is added into S for x*. This guarantees that $W_x \cap S$ will be nonempty if W_x is infinite, and the condition $y \geq 2x$ guarantees that $\overline{S}$ will be infinite.

Formally, we define a partial recursive function f such that $f(x)$ is the element enumerated in S for x (if such an element exists), namely

$$f(x) = ((\mu z)[T(z.0, x, z.1) \,\&\, z.1 \geq 2x]).1.$$

This definition does indeed guarantee that $f(x) \geq 2x$ and $f(x) \in W_x$, if such an element exists, otherwise $f(x)\uparrow$. Define $S = \mathrm{ran}\, f$. If W_x is infinite, then W_x contains some element $\geq 2x$, which implies that $f(x)\downarrow$, hence $f(x) \in S \cap W_x$. Since this conclusion holds for every x, it follows that no W_x can be an infinite r.e. subset of $\overline{S}$. On the other hand, S contains for every z at most z elements $< 2z$, namely $f(x), \ldots, f(z-1)$, hence $|\overline{S} \cap \{0, \ldots, 2z-1\}| \geq z$ for every z, hence $\overline{S}$ is infinite. This proves that $\overline{S}$ is immune and S is simple. □

We have just established the first nontrivial result on the structure of r.e. sets under $\leq_m$ and $\leq_1$. Some further results can also be shown by using simple sets in an appropriate way.

Theorem 6.17: (i) If $A \leq_1 B$ and B is immune (simple), then A is finite or immune (cofinite or simple).

(ii) There exist two r.e. sets A, B such that $A \leq_m B$, but $A \not\leq_1 B$.

(iii) There exists an r.e. set A such that A is neither recursive, nor 1-complete, nor simple.

Proof: (i) Let $A \leq_1 B$ via f. Assume first that B is immune. Then A cannot contain an infinite r.e. set C, otherwise $f(C)$ would be an infinite r.e. subset of B. Hence A is finite or immune. If on the other hand B is simple, then $\overline{A} \leq_1 \overline{B}$ by theorem 5.4 (i), hence $\overline{A}$ is finite or immune, hence A is simple or cofinite, since A is also r.e. by theorem 5.5 (ii).

(ii) Let B be the set S from theorem 6.16, and let $A = B \oplus \emptyset$. Then A contains the infinite r.e. set $\{\, 2x+1 \mid x \in \omega \,\}$, hence $A \not\leq_1 B$ by (i). On the other hand, let b be some fixed element of $\overline{B}$. Then $A \leq_m B$ via f, where

$$f(x) = \begin{cases} x/2 & \text{if } x \text{ is even,} \\ b & \text{if } x \text{ is odd.} \end{cases}$$

Indeed, if x is even, then $x \in A$ iff $x/2 \in B$, and if x is odd, then $x \notin A$.

(iii) Let A be as in (ii). Clearly, we have not only $A \leq_m B$, but also $B \leq_m A$ via $\lambda x.2x$. Since A belongs to the same m-degree as a simple set, A is neither recursive nor m-complete. Also, it was already observed in (ii) that A is not simple. □

These results provide answers for some of the questions raised in section 6.2, before the consideration of joins. Concerning the remaining ones, we provide the following information without proofs.

It will established in (DCC) that there exist simple sets which are T-complete. This proves that the properties *T-complete* and *m-complete* are not equivalent, and that the relation $\leq_T$ and $\leq_m$ do not coincide. In fact, Post's simple set from theorem 6.16 is already T-complete, though we will not prove this fact. Finally, the most important question, known as *Post's problem*, is whether or not there exist r.e. sets which are neither recursive nor T-complete. A positive answer will also be proven in (DCC).

We conclude these lecture notes with two more exercises. The last one deals with minimal upper bounds for pairs of r.e. 1-degrees, as promised after theorem 6.8.

Exercise 6.20: (i) [3] Show that if A and B are simple, then so are $A \cap B$ and $A \oplus B$.

(ii) [4] Show that there exists a set $A \subset \omega$ such that A and $\overline{A}$ are both immune.

Exercise 6.21: (i) [4] Let $A \subseteq \omega$, $a \in A$. Show that the following assertions are equivalent:

(a) A is neither finite nor immune.

(b) $A \leq_1 A - \{a\}$.

(c) $A \leq_1 B$ for some $B \subsetneq A$.

(ii) [6] Assume that $A \leq_1 D$ and $B \leq_1 D$. Show that there exists a set C such that

(a) $A \leq_1 C$, $B \leq_1 C$ and $C \leq_1 D$.

(b) If A and B are r.e., then C is r.e.

(c) If A and B are simple, then C is simple.

Conclude that if $\deg_1(C)$ is the least upper bound of $\deg_1(A)$ and $\deg_1(B)$, and A and B are r.e. (simple), then C is r.e. (simple).

(iii) [8] Construct two simple sets A, B such that $A \not\leq_1 B$ and $B \not\leq_1 A$, that is, A and B have incomparable 1-degrees.

(iv) [5] Let A, B be as in (iii). Show that there exists no minimal upper bound for $\deg_1(A)$ and $\deg_1(B)$. (Note the difference between least upper bound and minimal upper bound: $\mathbf{c}$ is the least upper bound of $\mathbf{a}$ and $\mathbf{b}$, iff every upper bound $\mathbf{d}$ of $\mathbf{a}$ and $\mathbf{b}$ satisfies $\mathbf{c} \leq \mathbf{d}$; but $\mathbf{c}$ is a minimal upper bound of $\mathbf{a}$ and $\mathbf{b}$, iff no upper bound $\mathbf{d}$ of $\mathbf{a}$ and $\mathbf{b}$ satisfies $\mathbf{d} < \mathbf{c}$. The existence of pairs of r.e. one-degrees without least upper bound was first shown by Young [28].)

References

The following list of references includes some textbooks which provide an introduction to the theory of computation (but none which only deal with special topics beyond the level of these lecture notes), and research papers which were cited at some place.

[1] Börger, E.: Berechenbarkeit, Komplexität, Logik. Vieweg, 1985.

[2] Brainerd, W.S., Landweber, L.H.: Theory of Computation. Wiley, 1974.

[3] Buchberger, B., Roider, B.: Certain decompositions of Gödel numberings and the semantics of programmins languages. In: Proceedings of the International Symposium of Theoretical Programming, Novosibirsk 1972; Springer Lecture Notes 5, 1974.

[4] Cutland, N.J.: Computability: An Introduction to Recursive Function Theory. Cambridge University Press, 1980.

[5] Davis, M.D. (ed.): The Undecidable. Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions. Raven Press, 1965.

[6] Davis, M.D., Weyuker, E.J.: Computability, Complexity and Languages. Academic Press, 1983.

[7] Hennie, F.: Introduction to Computability. Addison-Wesley, 1977.

[8] Hermes, H.: Enumerability. Decidability. Computability. Springer-Verlag, 1969.

[9] Kleene, S.K.: General recursive functions of natural numbers. *Mathematische Annalen* **12** (1936), p. 727–742.

[10] Kleene, S.K.: On notations for ordinal numbers. *Journal of Symbolic Logic* **10** (1938), p. 150–155.

[11] Knuth, D.E.: The Art of Computer Programming. Vol. 1: Fundamental Algorithms. Addison-Wesley, 1973.

[12] Lachlan, A.H.: A note on universal sets. *Journal of Symbolic Logic* **31** (1966), p. 573–574.

[13] Machtey, M., Young, P.R.: An Introduction to the General Theory of Algorithms. North-Holland, 1978.

[14] Mal'cev, A.I.: Algorithms and Recursive Functions (English translation). Wolters-Noordhoff Publishing Company, 1970.

[15] McNaughton, R.M.: Elementary Computability, Formal Languages, and Automata. Prentice-Hall, 1982.

[16] Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice Hall, 1967.

[17] Odifreddi, P.: Classical Recursion Theory. North Holland, 1989.

[18] Owings, J.C.: Diagonalization and the recursion theorem. *Notre Dame Journal of Symbolic Logic* **14** (1973), p. 95–99.

[19] Péter, R.: Recursive Functions (third edition). Academic Press, 1967.

[20] Post, E.L.: Recursively enumerable sets of positive integers and their decision problems. *Bull. Amer. Math. Soc.* **50** (1944), p. 284–316.

[21] Rogers, H.: Theory of Recursive Functions and Effective Computability. McGraw Hill, 1967.

[22] Shepherdson, J.C., Sturgis, H.E.: Computability of Recursive Functions. *Journal of the ACM* **10** (1963), p. 217–255.

[23] Soare, R.I.: Recursively Enumerable Sets and Degrees. Springer-Verlag, 1987.

[24] Sommerhalder, R., van Westrhenen, S.C.: The Theory of Computability. Addison-Wesley 1988.

[25] Tourlakis, G.J.: Computability. Reston Publishing Company, 1984.

[26] Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* **42** (1936), p. 230–265.

[27] Weihrauch, K.: Computability. Springer-Verlag (EATCS Monographs on Theoretical Computer Science), 1987.

[28] Young, P.R.: On reducibility by recursive functions. *Proceeding of the American Mathematical Society* **115** (1965), p. 329–339.